Windows Presentation Foundation Task

April 2023

1 Introduction

The purpose of this document is to demonstrate a sample task you might encounter during Windows Forms graded laboratories. The first part of this guide will show you an example solution for the laboratory part.

2 Laboratory Task

2.1 Description

The task is to write a simple application allowing to manage contacts. Contact can be added in 2 ways: using DataGrid control or filling out a form in a new window. In the task archive, you should be able to find **ContactManagerLab.exe** file - an example app of the laboratory part of the task.

Requirements

- Main window:
  - minimum size 500 x 500, initial size 800 x 600
  - menu bar at the top of the window
  - the rest of the window is divided into tabs: list of contacts and grid for editing and adding new contacts
  - ”Contact Manager” written on the title bar of the window
  - window appears in the center of the screen
- Top menu:
  - ’File’ contains:
    - ’Import’ - inactive
    - ’Export’ - inactive
    - ’Exit’ - closes the application
  - ’Contacts’ contains:
    - ’Add contact’ - opens a new window allowing to add contact

        * 'Clear contacts' - clears the list of contacts
    - 'About' - displays a brief information about the application with an 'Information' icon

- Contact class contains the following information:
    - first name
    - last name
    - email address
    - phone number
    - gender

- Main part of the program:
    - window is divided into two tabs: list of contacts and grid for editing, adding, and deleting contacts
    - proper template must be created for displaying contact information - elements should be arranged as in the sample application
    - only first name and last name of the contact and an avatar corresponding to the gender should be displayed on the contact list
    - contact list must correspond to the content of the grid (binding)
    - changes made in the grid should be visible on the contact list immediately after being approved
    - even and odd positions have different background colors (in the sample application: #FFAFC5FF and #FF75A1FF)
    - implementation of the above functionality must be resistant to deleting elements from the middle, sorting, and other actions (it cannot be achieved only by setting the background of the element when adding it to the list)
    - contacts can be edited, added and deleted using the grid
    - the grid displays an additional row allowing to add a new contact

- Window for adding a new contact:
    - appears in the center of the screen
    - its appearance blocks and 'grays out' the main window
    - contains a form allowing to fill in contact details
    - ComboBox displaying values of the appropriate enum should be used for choosing gender
    - user can approve adding contact or cancel

- Notes:
    - the application is resistant to window scaling, controls adjust to changes in the window size
    - in case of ambiguity, the program must be consistent with the behavior and appearance of the sample application (with the exception of potential bugs)

Hints

- ObservableCollection
- CollectionViewSource
- ListBox.ItemContainerStyle, ListBox.ItemTemplate, DataTemplate, ListBox.ItemContainerStyle
- DataGrid
- Trigger, AlternationIndex

Grading criteria

- menu with working Exit and About buttons - 2 points
- list of contacts with proper template - 3 points
- different background colors for even and odd contacts - 1 point
- data grid - 2 points
- window for adding a new contact - 3 points
- clearing the list of contacts - 1 point

3 Solution

3.1 Creating a project

To begin, we will create a Visual Studio project using the WPF Application template:

1. Choose File → New → Project... → WPF Application (C#)
2. Name it however you like.
3. In Additional Information part set Framework to .NET 6 or .NET 7.
4. In the created app you will have the main window already created (called ”MainWindow”).

3.2 Main Window appearance

1. To add all the controls in this tutorial, we will use XAML markup. XAML (eXtensible Application Markup Language) is a declarative markup language used in Windows Presentation Foundation (WPF) to define the user interface of an application. Alternatively, you may also use the toolbox and drag-and-drop the controls onto the designer to accomplish this step.

   (a) To open the Designer window, double-click on ”MainWindow.xaml” in the Solution Explorer. If the Designer window is already open, you can skip this step.

   (b) To open the partial code of the MainWindow class, double-click on ”MainWindow.xaml.cs” in the Solution Explorer. This will open the code-behind file for the MainWindow.

2. Set the window title to a more appropriate one. You can change the window title in WPF by setting the **Title** property of the **Window** element in XAML.

3. The window should start in the center of the screen. To center a WPF window using XAML markup, you can set the **WindowStartupLocation** property to **CenterScreen** in the opening tag of the **Window** element.

4. One of the main window requirements is to set its minimum size to 500 x 500 and initial size to 800 x 600. To set an initial size for a WPF window in XAML, you can set the **Width** and **Height** properties, and to set minimum size **MinimumWidth** and **MinimumHeight** properties of the **Window** element to the desired values.

```
<Window x:Class="ContactManager.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:ContactManager"
        mc:Ignorable="d"
        Title="Contact Manager" Height="600" Width="800"
        MinWidth="500" MinHeight="500" WindowStartupLocation="CenterScreen">
    <Grid>

    </Grid>
</Window>
```

5. Now we will setup the window layout:

   (a) In the already existing **Grid** element we will add row definitions. Define two rows with width **Auto** and $*$. **Auto** width means that the size of the row or column will be automatically determined based on the size of its content. For example, if you have a Grid with a row that contains a TextBox, setting the height of the row to "**Auto**" will cause the row to be sized to fit the height of the TextBox. The $*$ character represents a weighted proportion of available space. In this case the second row will consume all remaining available space.

   ```
   <Grid.RowDefinitions>
       <RowDefinition Height="Auto"/>
       <RowDefinition Height="*"/>
   </Grid.RowDefinitions>
   ```

   (b) A menu bar will be positioned in the top row. To accomplish this, we will insert a **Menu** element into the existing **Grid** element and specify its structure and on click events. Property **Grid.Row** is used to select the position in the grid of our Menu control. You should click on the event name and press **F12** (go to definition) to generate definition of the event inside the partial class.

```
 <Menu Grid.Row="0">
    <MenuItem Header="File">
        <MenuItem Header="Import" IsEnabled="False" />
        <MenuItem Header="Export" IsEnabled="False" />
        <Separator />
        <MenuItem Header="Exit" Click="MenuItem_Exit" />
    </MenuItem>
    <MenuItem Header="Contacts">
        <MenuItem Header="Add contact" Click="MenuItem_AddContact" />
        <MenuItem Header="Clear contacts" Click="MenuItem_ClearContacts" />
    </MenuItem>
    <MenuItem Header="About" Click="MenuItem_About" />
</Menu>
```

(c) In the remaining space of our **Grid** we will place a **TabControl** with two **TabItems** with List and Grid headers containing **ListBox** and **DataGrid** respectively.

```
 <TabControl Grid.Row="1">
    <TabItem Header="List">
        <ListBox/>
    </TabItem>
    <TabItem Header="Grid">
        <DataGrid/>
    </TabItem>
</TabControl>
```

3.3 Data classes

1. Create a class that will hold the contact information and an enum for gender. To do this, add a new item to the project called ”Contact.cs” and create the following classes:

```
public class Contact
{
    public string? Name { get; set; }
    public string? Surname { get; set; }
    public string? Email { get; set; }
    public string? Phone { get; set; }
    public Gender Gender { get; set; }

    public Contact()
    {
    }
}

public enum Gender
{
    Male,
    Female
}
```

2. In the partial class create a collection that will store contact information in our application. We will use **ObservableCollection** for this purpose, as it provides automatic notifications when the collection changes. Initialize it in the constructor. We will also asign the Contacts list to the **DataSource**. This will allow us to bind the Contacts collection to the **ListBox** and **DataGrid** control in the XAML.

```
 public partial class MainWindow : Window
{
    private ObservableCollection<Contact> Contacts { get; set; }

    public MainWindow()
    {
        InitializeComponent();

        Contacts = new ObservableCollection<Contact>();

        DataContext = Contacts;
    }

    ...
}
```

3.4 Add new contact window

To create a new window for adding a contact, follow these steps:

1. First add new item to the project. This time its type should be **"Window (WPF)"**. Choose a name for new item, such as AddContactWindow.xaml.

2. Open the designer of the new window by double-clicking on the AddContactWindow.xaml file in the Solution Explorer. This will open the XAML designer where you can design the layout of the window.

3. Add the following properties to the Window element:

   (a) Title="Add contact": This sets the title of the window to "Add contact".

   (b) Height="250": This sets the height of the window to 250 pixels.

   (c) Width="350": This sets the width of the window to 350 pixels.

   (d) WindowStartupLocation="CenterScreen": This sets the startup location of the window to the center of the screen.

   (e) ResizeMode="NoResize": This disables the ability for the user to resize the window.

   (f) ShowInTaskbar="False": This hides the window from the taskbar.

   (g) WindowStyle="None": This removes the window's border and title bar.

   (h) BorderBrush="Black": This sets the color of the window's border to black.

   (i) BorderThickness="3": This sets the thickness of the window's border to 3 pixels.

4. To create the layout and structure of a window for adding a new contact, we will divide the grid into three rows. Add the following XAML code inside the Window element to define the row definitions:

```
 <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="Auto"/>
</Grid.RowDefinitions>
```

   This will create three rows with the first and last row having a fixed height and the middle row occupying all the remaining space.

5. To add a label that displays the title of the window, you can add a Label element to the first row of the grid and set its content to "Add new contact".

```
<Label Grid.Row="0" HorizontalAlignment="Center" FontSize="24">Add new contact</Label>
```

6. The second row of the grid will contain a nested grid that has two columns and five rows. Each row will contain a label and a text box or combo box, which correspond to the fields of the contact that can be added. The text boxes will be later bound to the properties of a "NewContact" object, which will be defined in the partial class.

```
<Grid Grid.Row="1">
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Label Grid.Column="0" Grid.Row="0">Name:</Label>
    <Label Grid.Column="0" Grid.Row="1">Surname:</Label>
    <Label Grid.Column="0" Grid.Row="2">Email:</Label>
    <Label Grid.Column="0" Grid.Row="3">Phone:</Label>
    <Label Grid.Column="0" Grid.Row="4">Gender:</Label>
    <TextBox Grid.Column="1" Grid.Row="0" Height="24"/>
    <TextBox Grid.Column="1" Grid.Row="1" Height="24"/>
    <TextBox Grid.Column="1" Grid.Row="2" Height="24"/>
    <TextBox Grid.Column="1" Grid.Row="3" Height="24"/>
    <ComboBox Grid.Column="1" Grid.Row="4" Height="24"/>
</Grid>
```

7. In the third row of the grid, add a **StackPanel** with a horizontal orientation and set its HorizontalAlignment property to Center. Inside the StackPanel, create two buttons. The first button should have a label "Add contact" and be associated with a click event named "AddContact". The second button should have a label "Cancel" and be associated with a click event named "Cancel". To create the event handlers, you can use the **F12** shortcut. Set the Width and Height properties of both buttons to 100 and 30 respectively. Additionally, to separate the two buttons, set the Margin property of the "Cancel" button to "15,0,0,0".

```
 <StackPanel Grid.Row="2" Orientation="Horizontal" HorizontalAlignment="Center">
    <Button Width="100" Height="30" Click="AddContact">Add contact</Button>
    <Button Width="100" Height="30" Click="Cancel" Margin="15,0,0,0">Cancel</Button>
</StackPanel>
```

8. We now need to add a Contact field to the partial class associated with the AddContactWindow. In the constructor of this partial class, we will create a new instance of the Contact class and assign it to the NewContact property. This property will be used for bindings to the text boxes in the window. We will also set the DataContext property of the window to the NewContact property, so we can bind it to the textboxes later.

```
public partial class AddContactWindow : Window
{
    public Contact NewContact { get; private set; }

    public AddContactWindow()
    {
        InitializeComponent();
        NewContact = new Contact();
        DataContext = NewContact;
    }
}
```

9. Bind the fields of the NewContact object to the text boxes and combo boxes in the XAML markup. Set the Text property of each TextBox to the appropriate binding expression:

```
<TextBox Grid.Column="1" Grid.Row="0" Height="24"
Text="{Binding Name, UpdateSourceTrigger=PropertyChanged}"/>
<TextBox Grid.Column="1" Grid.Row="1" Height="24"
Text="{Binding Surname, UpdateSourceTrigger=PropertyChanged}"/>
<TextBox Grid.Column="1" Grid.Row="2" Height="24"
Text="{Binding Email, UpdateSourceTrigger=PropertyChanged}"/>
<TextBox Grid.Column="1" Grid.Row="3" Height="24"
Text="{Binding Phone, UpdateSourceTrigger=PropertyChanged}"/>
```

This will enable the data entered into the text boxes to be stored in the corresponding properties of the NewContact object. Note that the UpdateSourceTrigger=PropertyChanged attribute ensures that the data is immediately updated in the object as soon as the user types in the text box.

10. To bind to Gender field we need to create a list of possible enum values. We will accomplish this using only XAML markup. First we will add System namespace in our xaml file to get access to the Enum class. In the Window element add the following property:

```
xmlns:System="clr-namespace:System;assembly=mscorlib"
```

11. Now we need to create a list of possible enum values. We will create it using **ObjectDataProvider** in **Window.Resources** element. **Window.Resources** is a XAML element that defines a collection of resources that can be used by the elements within a Window or UserControl. Resources can include objects such as styles, templates, brushes, and data bindings. By defining resources in the **Window.Resources** section, they can be easily shared across different controls within the window. The **ObjectDataProvider** allows you to bind to an object and its properties, methods, and events. In this case, it is being used to expose the values of the **Gender** enum defined in the local namespace. The **MethodName** property is set to **"GetValues"**, which returns an array of the values of the enumeration.

```
<Window.Resources>
    <ObjectDataProvider x:Key="GenderList" MethodName="GetValues"
                        ObjectType="{x:Type TypeName=System:Enum}" >
        <ObjectDataProvider.MethodParameters>
            <x:Type TypeName="local:Gender"/>
        </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>
</Window.Resources>
```

12. Then we will bind the ComboBox **ItemSource** to the created **GenderList**, and **SelectedItem** to the **Gender** Field in the **NewContact**.

```
<ComboBox Grid.Column="1" Grid.Row="4" Height="24"
          ItemsSource="{Binding Source={StaticResource GenderList}}"
          SelectedItem="{Binding Path=Gender}" />
```

13. Now we will implement AddContact and Cancel event handlers. In order to make the "Add Contact" and "Cancel" buttons functional, we need to implement their associated methods, that will set the **DialogResult** of the window to true and false, respectively, and then close the window using the **Close()** method.

```
private void AddContact(object sender, RoutedEventArgs e)
{
    DialogResult = true;
    Close();
}

private void Cancel(object sender, RoutedEventArgs e)
{
    DialogResult = false;
    Close();
}
```

3.5 Top Menu

1. In the partial class of MainWindow you should already have auto generated event handlers: **MenuItem_AddContact**, **MenuItem_ClearContacts**, **MenuItem_Exit**, **MenuItem_About**.

2. In the event handler **MenuItem_AddContact** we will create a new instance of the AddContactWindow and use **ShowDialog** method that displays the window as a modal dialog box. It means that the user cannot interact with any other windows in the application until the dialog window is closed. If the user clicks the Add Contact button, **ShowDialog** returns true, and we add the new contact to the Contacts collection. If the user clicks the Cancel button, it returns false, as defined in the previous section. In between, we will change the opacity of the main window to gray out the window while the dialog is open.

```
private void MenuItem_AddContact(object sender, RoutedEventArgs e)
{
    Opacity = 0.5;

    var addContactWindow = new AddContactWindow();
    if (addContactWindow.ShowDialog().Value)
    {
        Contacts.Add(addContactWindow.NewContact);
    }

    Opacity = 1;
}
```

3. In the **MenuItem_ClearContacts** event handler we simply clear the **Contacts** list:

```
private void MenuItem_ClearContacts(object sender, RoutedEventArgs e)
{
    Contacts.Clear();
}
```

4. In the **MenuItem_Exit** event handler we simply call the **Close()** method:

```
private void MenuItem_Exit(object sender, RoutedEventArgs e)
{
    Close();
}
```

5. In the **MenuItem_About** event handler we call the "MessageBox.Show" method which displays a message box with the text "This is a simple contact manager." The message box also has a title of "Contact Manager", an OK button, and an information icon:

```
private void MenuItem_About(object sender, RoutedEventArgs e)
{
    MessageBox.Show("This is a simple contact manager.",
    "Contact Manager", MessageBoxButton.OK,
    MessageBoxImage.Information);
}
```

3.6 ListBox and DataGrid

1. We start by copying Resources directory included in the zip file containing man.png and woman.jpg files to our project. We need to display them in our ListView. Change the **Build Action** of the two images to **Resource**.

   (a) Right-click on the image file in the Solution Explorer.

   (b) Select "Properties" from the context menu.

   (c) In the Properties window, find the "Build Action" setting under the "Advanced" section.

   (d) Click on the drop-down menu next to "Build Action" and select "Resource".

   The image will be now embedded as a resource in the assembly. This means that you can access the image as a resource within your code, without needing to distribute the image as a separate file.

2. We will add the images now to Window.Resources element in MainWindow.xaml. Adding images to Window.Resources allows you to reference them in your XAML markup throughout the window, making it easier to use them repeatedly. By doing this, you can centralize your resources in one place and avoid redundancy in your XAML code.

```
<Window.Resources>
    <BitmapImage x:Key="WomanAvatar" UriSource="/Resources/woman.jpg" />
    <BitmapImage x:Key="ManAvatar" UriSource="/Resources/man.png" />
</Window.Resources>
```

3. We will define a **DataTemplate** with the key "ContactListItemTemplate" that will be used to display each item in a **ListBox**. The template consists of a **Grid** with two columns: the first column contains a **StackPanel** that displays the Name and Surname properties of a Contact object, and the second column contains a **Border** that contains an **Image** element.

   The **Image** element's **Source** property is set using a **DataTrigger** based on the Gender property of the Contact object. If the Gender is "Female", the Image will display the image resource with the key "WomanAvatar", which has been added to the **Window.Resources** as a **BitmapImage** object. If the Gender is "Male", the Image will display the image resource with the key "ManAvatar".

```
<DataTemplate x:Key="ContactListItemTemplate" DataType="ListBoxItem">
    <Grid HorizontalAlignment="Stretch">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="Auto"/>
        </Grid.ColumnDefinitions>
        <StackPanel Grid.Column="0" Orientation="Horizontal"
        HorizontalAlignment="Stretch" VerticalAlignment="Center">
            <Label Content="{Binding Name}" FontSize="20" Padding="3,0,0,0"/>
            <Label Content="{Binding Surname}" FontSize="20" Padding="3,0,0,0" />
        </StackPanel>
        <Border Grid.Column="2" VerticalAlignment="Center"
        Width="50" Height="50" BorderBrush="Black" BorderThickness="1" Margin="3">
            <Image Stretch="Fill" >
                <Image.Style>
                    <Style TargetType="{x:Type Image}">
                        <Style.Triggers>
                            <DataTrigger Binding="{Binding Gender}" Value="Female">
                                <Setter Property="Source"
                                Value="{StaticResource WomanAvatar}"/>
                            </DataTrigger>
                            <DataTrigger Binding="{Binding Gender}" Value="Male">
                                <Setter Property="Source"
                                Value="{StaticResource ManAvatar}"/>
                            </DataTrigger>
                        </Style.Triggers>
                    </Style>
                </Image.Style>
            </Image>
        </Border>
    </Grid>
</DataTemplate>
```

4. Inside **Window.Resources** we will also define a CollectionViewSource. It provides a view on top of the collection, enabling you to to sort, filter, group, and manipulate the data in a collection without modifying the underlying data source.

```
<CollectionViewSource Source="{Binding}"
                      x:Key="ContactsViewSource"/>
```

5. Now we will modify our **ListBox** control. We will set its **ItemsSource** property to bind it to the previously defined **ContactsViewSource**. We also need to set the **ItemTemplate**

property to the **DataTemplate** resource named ContactListItemTemplate, which defines how each item in the list will be displayed. The **HorizontalContentAlignment** property is set to **Stretch**, which means that the content of each ListBox item will fill the entire horizontal width of the ListBox.

```
<ListBox ItemsSource="{Binding
                       Source={StaticResource ResourceKey=ContactsViewSource}}"
         ItemTemplate="{StaticResource ContactListItemTemplate}"
         HorizontalContentAlignment="Stretch"/>
```

6. Now we define a style inside **Window.Resources** named ”ContactListItemStyle” which targets **ListBoxItem**. Inside the style, there are two triggers that set the background color of the **ListBoxItem** based on its ”ItemsControl.AlternationIndex” property. The **AlternationIndex** property is used to apply a different background color to alternate items in the ListBox to make them more distinguishable. In this example, items with an AlternationIndex of 0 have a background color of ”#FFAFC5FF” and items with an AlternationIndex of 1 have a background color of ”#FF75A1FF”.

```
<Style x:Key="ContactListItemStyle" TargetType="ListBoxItem">
    <Style.Triggers>
        <Trigger Property="ItemsControl.AlternationIndex"
        Value="0">
            <Setter Property="Background" Value="#FFAFC5FF" />
        </Trigger>
        <Trigger Property="ItemsControl.AlternationIndex"
        Value="1">
            <Setter Property="Background" Value="#FF75A1FF" />
        </Trigger>
    </Style.Triggers>
</Style>
```

7. Now we apply previously defined style by setting ItemContainerStyle and AlternationCount properties inside our ListView element.

```
<ListBox ItemsSource="{Binding}"
         ItemTemplate="{StaticResource ContactListItemTemplate}"
         HorizontalContentAlignment="Stretch"
         ItemContainerStyle="{StaticResource ContactListItemStyle}"
         AlternationCount="2"/>
```

8. We will now modify our **DataGrid** to bind it to the Contacts collection. We can auto generate columns using **AutoGenerateColumns** property. To allow the user to add new items to the DataGrid set the **CanUserAddRows** to True.

```
<DataGrid ItemsSource="{Binding}" AutoGenerateColumns="True"
          CanUserAddRows="True" />
```

4 Home Part

Requirements

- Top menu:
  - 'File' contains:
    - * 'Import' - an active button for importing contacts from an XML file
    - * 'Export' - active button for exporting contacts to an XML file
- "Letter" tab:
  - selecting any contact changes the template and displays its additional information: phone number and e-mail address
  - the background for the selected contact should be #FFFDCF6C
  - hovering over a contact should not highlight it (as in the sample app)
  - after right-clicking on a contact, a context menu with the "Delete contact" option is displayed
  - clicking on "Delete contact" deletes the contact
- "Validation settings" tab:
  - by default its content is locked
  - the user can unlock the content by clicking on the "Unlock validation settings" button
  - the mechanism of converters should be used to block/unblock the visibility of the content of this tab
  - after unlocking the content of the tab, the user can assign different validation rules to each of the text fields contained in the window for adding a contact
  - after the selection of validation rules is completed, the user has the option to block the content of the tab again
- Registration window:
  - ensure the use of user-selected validation rules for individual form fields
  - in the event of an error, the frame of the field turns red, and the appropriate icon is displayed next to it (see example application)
  - after moving the mouse over the error icon, information about the error is displayed (in the form of a tooltip)
- Validation mechanism:
  - 3 validation rules should be implemented:
    - * a rule that checks if the minimum length of the entered text is ≥ 5 characters
    - * email address validation rule
    - * a rule that checks that the phone number is in the format XXX-XXX-XXX

  – the user should be able to add/remove validation rules without having to recompile the project, by adding/removing DLL files accordingly
  – the validation rule should be implemented as a class compiled to a DLL file, implementing the ValidationRule base class and the following interface:

```
public interface IValidation
{
    string Name { get; }

    string Description { get; }
}
```

  – in the case of correct implementation of the ValidationRule base class and the IValidation interface, it should be possible to exchange files with validation rules between students
  – below is an example implementation of a validation rule:

```
public class PhoneValidationRule : ValidationRule, IValidation
{
    public string Name => "Rule Name";

    public string Description => "Error description";

    public override ValidationResult Validate(object value, CultureInfo cultureInfo)
    {
        // TODO: Some validations

        return ValidationResult.ValidResult;
    }
}
```

Additional requirements

- it is mandatory to meet all the requirements of the first part of the task
- care should be taken that the controls behave and look like in the sample application (be careful with scaling, changing content, etc.)
- the format of the saved files should be human readable
- the visibility of individual areas should be set cleverly (using a custom converter)
- in case of ambiguities, the program must be consistent with the behavior and appearance of the sample application (except for potential bugs)

Hints

- INotifyPropertyChanged
- XMLSerializer

- Validation.ErrorTemplate, GetBindingExpression, ValidationRules
- Converter
- Build Path in Project properties

Punctation

- import/export of contacts to an XML file - 2 points
- contact details after selecting it on the list by clicking - 2 points
- validation implementation, without the need to separate them as separate DLL files - 0.5 points for each:
  - minimum length of 5 characters
  - e-mail adress
  - telephone number in the format XXX-XXX-XXX
- implementing the IValidation interface and adding validation rules as separate DLL files - 2 points
- adding the 'Validation settings' tab, which allows you to attach different validation rules to individual fields - 2 points
- implementation of blocking 3rd tab using converters - 1.5 points
- context menu and contact deletion - 1 point