

Введение

Платформа *Microsoft .Net* обеспечивает возможность разработки программных приложений на языках *C#*, *Visual Basic*, *C++* и *Java Script*. Для визуализации приложений используются языки разметки *XAML* и *HTML*. В данном лабораторном практикуме рассматриваются вопросы разработки программных приложений на языке программирования *C#* с использованием языка разметки *XAML*.

Язык расширенной разметки для приложений *XAML* и язык программирования *C#*, технологии *Windows Presentation Foundation (WPF)* являются основополагающими направлениями развития информационных технологий компании *Microsoft*. Сфера использования данных технологий стремительно расширяется. Многие разработчики во всем мире берут на вооружение именно эти технологии. Это определяет важность вопросов изучения данных технологий в высшей школе. Возможности языка *XAML*, технологии *WPF* эффективно поддерживаются инструментальной средой *Visual Studio*. Большое значение имеет локализация, как среды разработки, так и программных инструментов для русского языка. При проектировании приложения имеется возможность создавать насыщенные, высокоэффективные многоуровневые, интероперабельные программные системы. Освоение технологий проектирования программных систем с использованием *XAML* и *WPF* требует немалых усилий, но основные приемы дизайна, проектирования, кодирования и тестирования могут быть освоены достаточно быстро.

Данный лабораторный практикум ориентирован на закрепление теоретических основ и получение практических навыков по разработке и сопровождению программных систем с использованием технологии *WPF*. Технология разработки программных систем в лабораторном практикуме иллюстрируется детальным разбором практических примеров, приводятся *XAML*-описания компонентов систем, коды на языке *C#* и экранные формы, которые используются как при проектировании, так и при представлении результатов работы систем.

Представленные в лабораторном практикуме примеры разрабатывались для *Visual Studio 2017*, *Visual C# 2017*, *Net.Framework 4.7*, *SQL Server 2014*.

ЛАБОРАТОРНАЯ РАБОТА 1. Разработка моделей данных

Цель работы: Создать в проекте WPF модели данных.

Общие сведения

Создается WPF-приложение для обработки данных по сотрудникам компании. Приложение будет работать с данными по сотрудникам, представленные двумя классами (рисунок 1.1).

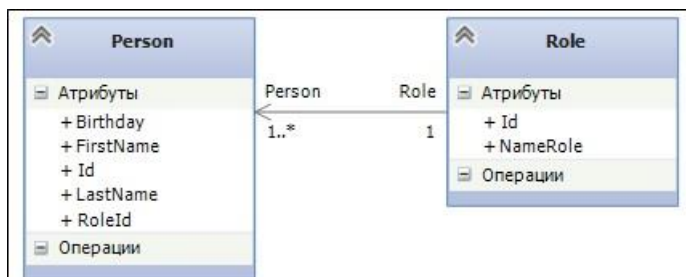


Рисунок 1.1 – классы, описывающие предметную область

Приложение должно отображать данные по сотрудникам и должностям в компании, редактировать, добавлять и удалять данные.

Создание проекта приложения

Создаем проект приложения на основе шаблона классического приложения на C#, указав имя приложения (рисунок 1.2).

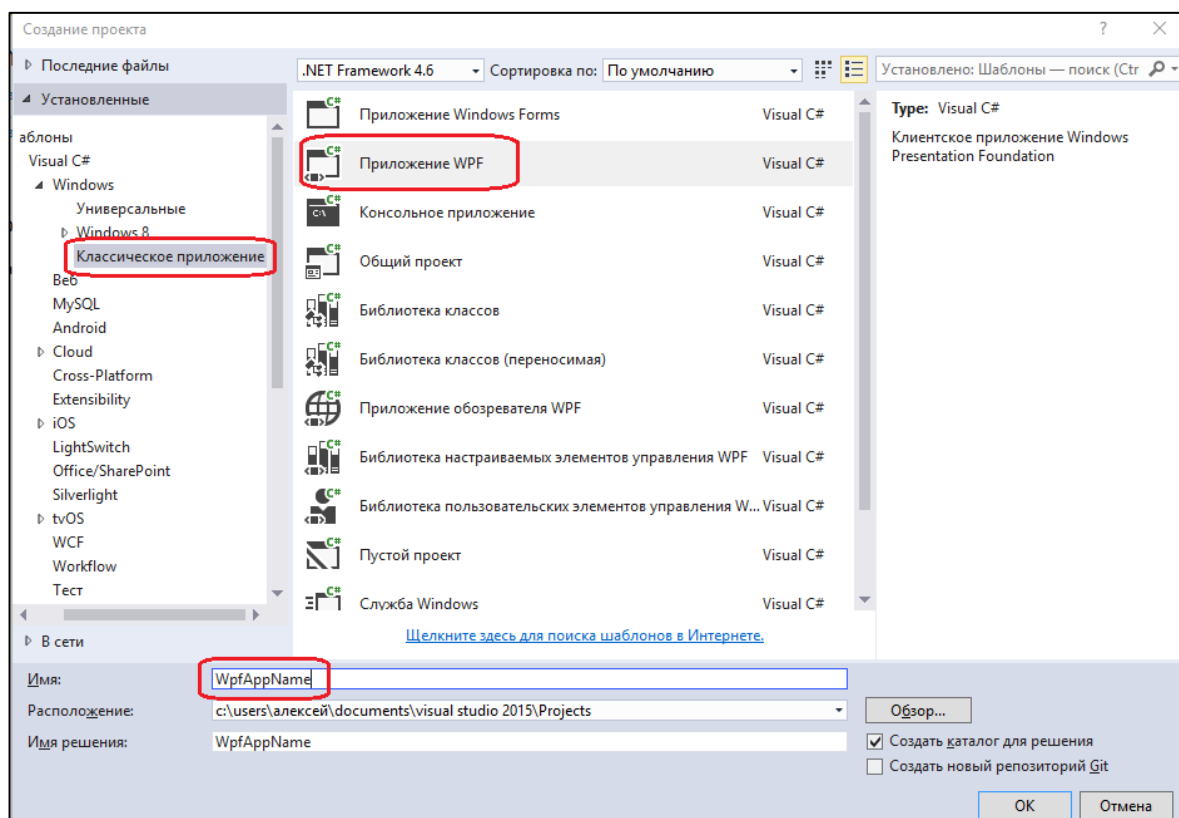


Рисунок 1.2 – создание приложения

В созданный проект добавим три папки: Model, View и ViewModel (рисунок 1.3)

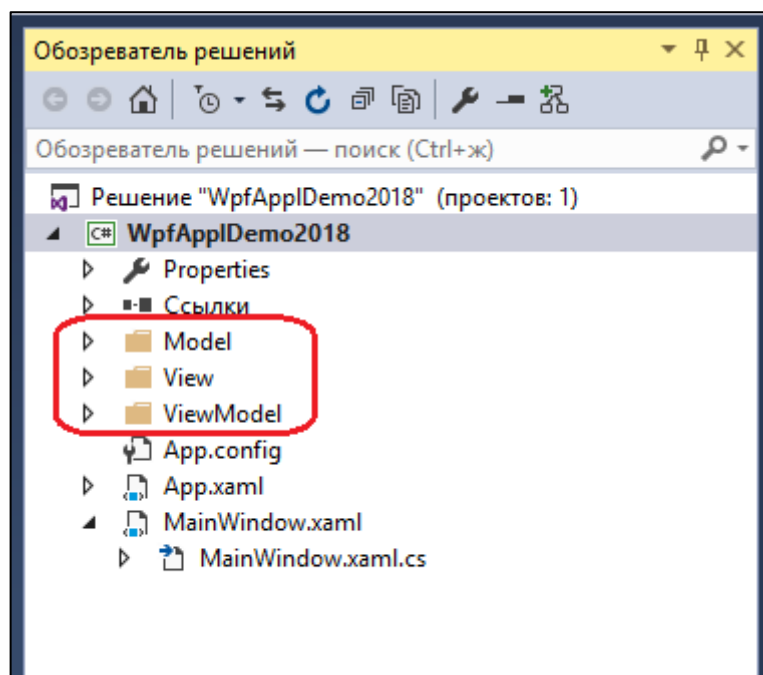


Рисунок 1.3 – Добавление папок

В папке Model создаем классы Person и Role, которые описывают предметную область (рисунок 1.4).

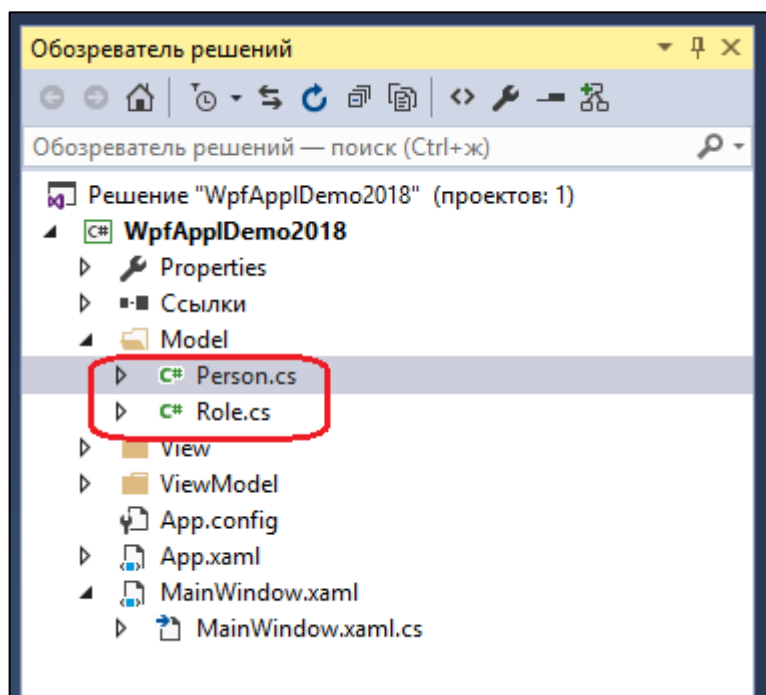


Рисунок 1.4 – добавление классов предметной области

Листинги классов Person и Role

```
using System;
namespace WpfApplDemo2018.Model
{
    public class Person
    {
        public int Id { get; set; }
        public int RoleId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime Birthday { get; set; }
        public Person() { }
        public Person(int id, int roleId, string firstName,
            string lastName, DateTime birthday)
        {
            this.Id = id;
            this.RoleId = roleId;
            this.FirstName = firstName;
            this.LastName = lastName;
            this.Birthday = birthday;
        }
    }
}
```

```
namespace WpfApplDemo2018.Model
{
    public class Role
    {
        public int Id { get; set; }
        public string NameRole { get; set; }
        public Role() { }
        public Role(int id, string nameRole)
        {
            this.Id = id;
            this.NameRole = nameRole;
        }
    }
}
```

В папке ViewModel создадим классы PersonViewModel и RoleView-Model, в которых будут сформированы коллекции данных по сотрудникам и должностям (рисунок 1.5)

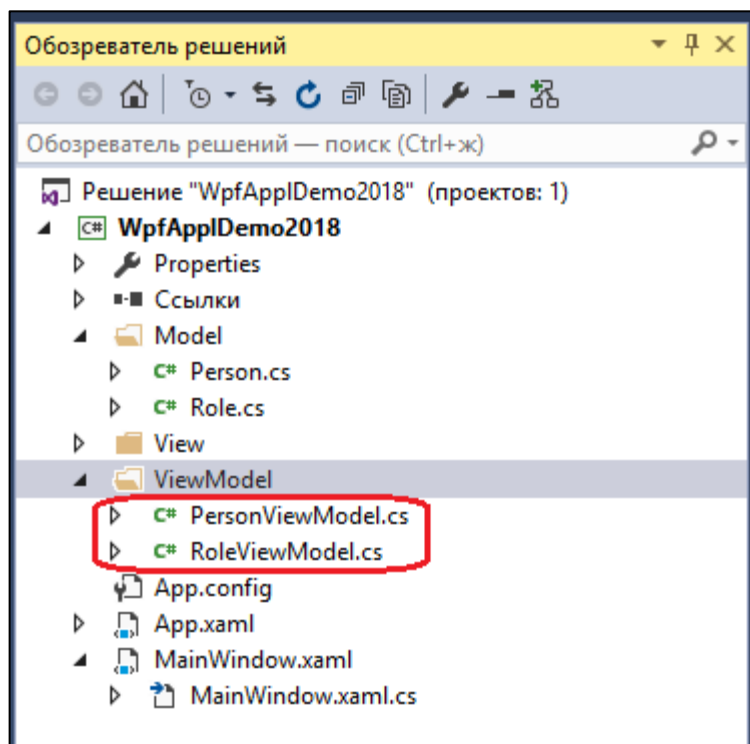


Рисунок 1.5 – формирование классов модели представления
Листинги классов PersonViewModel и RoleViewModel

```
using System;
using System.Collections.ObjectModel;
using WpfApp1Demo2018.Model;
namespace WpfApp1Demo2018.ViewModel
{
    public class PersonViewModel
    {
        public ObservableCollection<Person> ListPerson { get; set; } =
            new ObservableCollection<Person>();
        public PersonViewModel()
        {
            this.ListPerson.Add(
                new Person
                {
                    Id = 1,
                    RoleId = 1,
                    FirstName = "Иван",
                    LastName = "Иванов",
                    Birthday = new DateTime(1980, 02, 28)
                });
            this.ListPerson.Add(
                new Person
                {
                    Id = 2,
                    RoleId = 2,
                    FirstName = "Петр",
                    LastName = "Петров",
                    Birthday = new DateTime(1981, 03, 20)
                });
        }
    }
}
```

```

this.ListPerson.Add(
    new Person
    {
        Id = 3,
        RoleId = 3,
        FirstName = "Виктор",
        LastName = "Викторов",
        Birthday = new DateTime(1982, 04, 15)
    });
this.ListPerson.Add(
    new Person
    {
        Id = 4,
        RoleId = 3,
        FirstName = "Сидор",
        LastName = "Сидоров",
        Birthday = new DateTime(1983, 05, 10)
    });}}}

```

```

namespace WpfApplDemo2018.ViewModel
{
    public class RoleViewModel
    {
        public ObservableCollection<Role> ListRole { get; set; } = new
        ObservableCollection<Role>();
        public RoleViewModel()
        {
            this.ListRole.Add(new Role
            {
                Id = 1,
                NameRole = "Директор"
            });
            this.ListRole.Add(new Role
            {
                Id = 2,
                NameRole = "Бухгалтер"
            });
            this.ListRole.Add(new Role
            {
                Id = 3,
                NameRole = "Менеджер"
            });
        }
    }
}

```

В папке View создадим окна для представления данных о сотрудниках и должностях (рисунок 1.6).

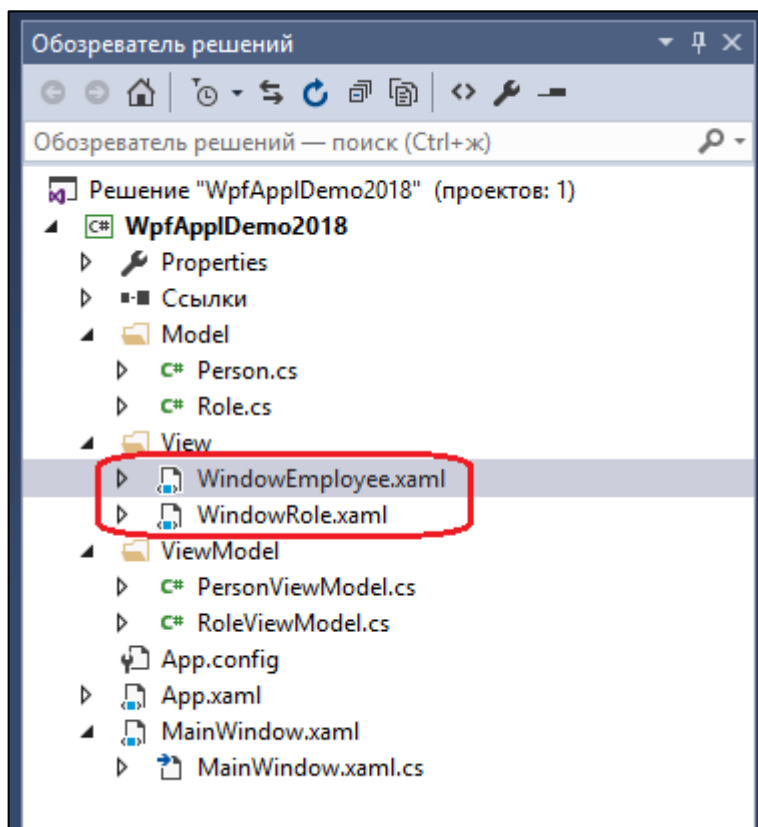


Рисунок 1.6 – создание оконных форм

Для представления данных по сотрудникам шаблон экранной формы будет иметь вид, представленный на рисунке 1.7, а для должностей – рисунок 1.8.

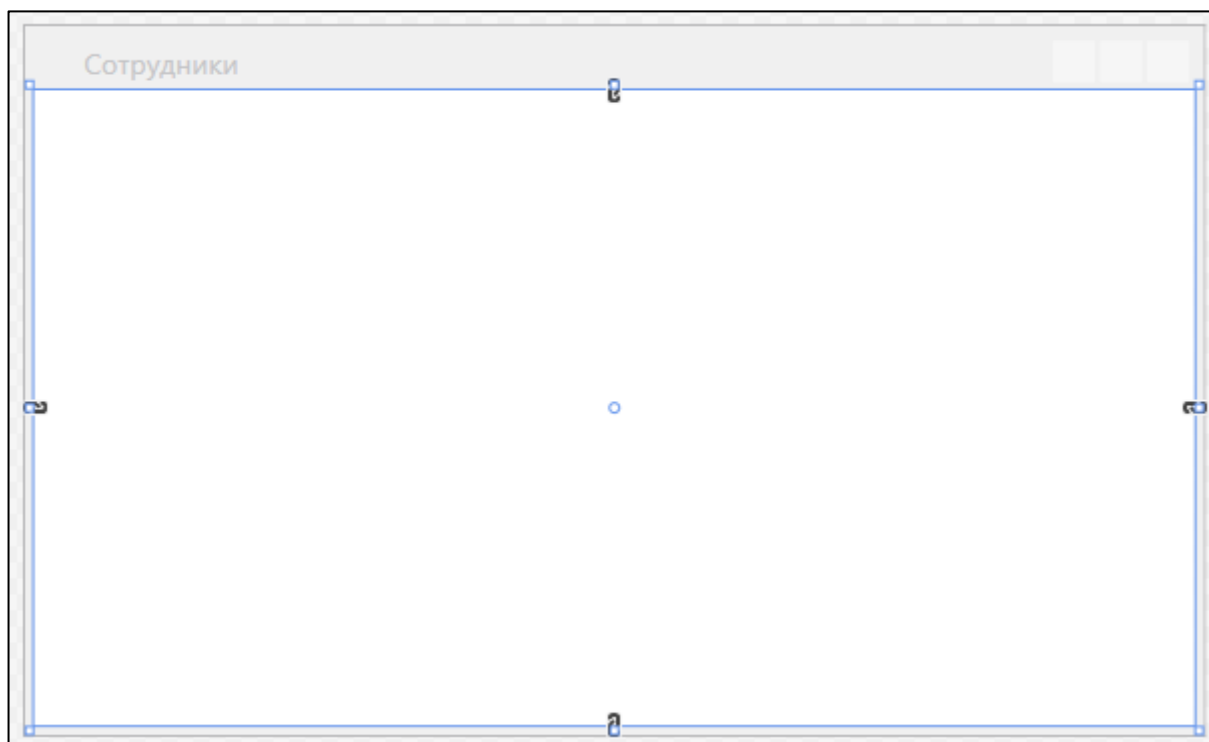


Рисунок 1.7 – шаблон экранной формы для сотрудников

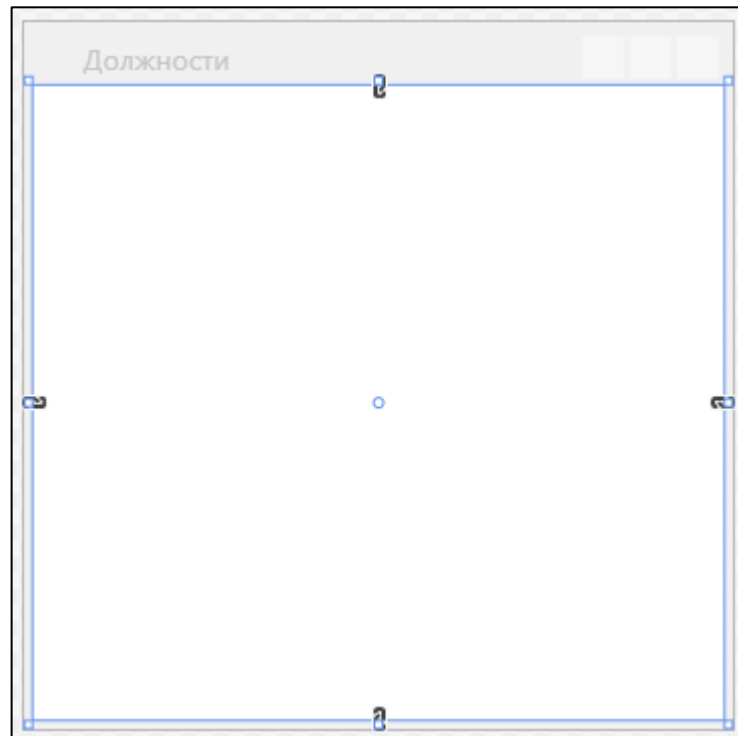


Рисунок 1.8 – шаблон экранной формы для должностей

На главном окне приложения создадим меню для вывода и обработки данных по сотрудникам и должностям (рисунок 1.9).

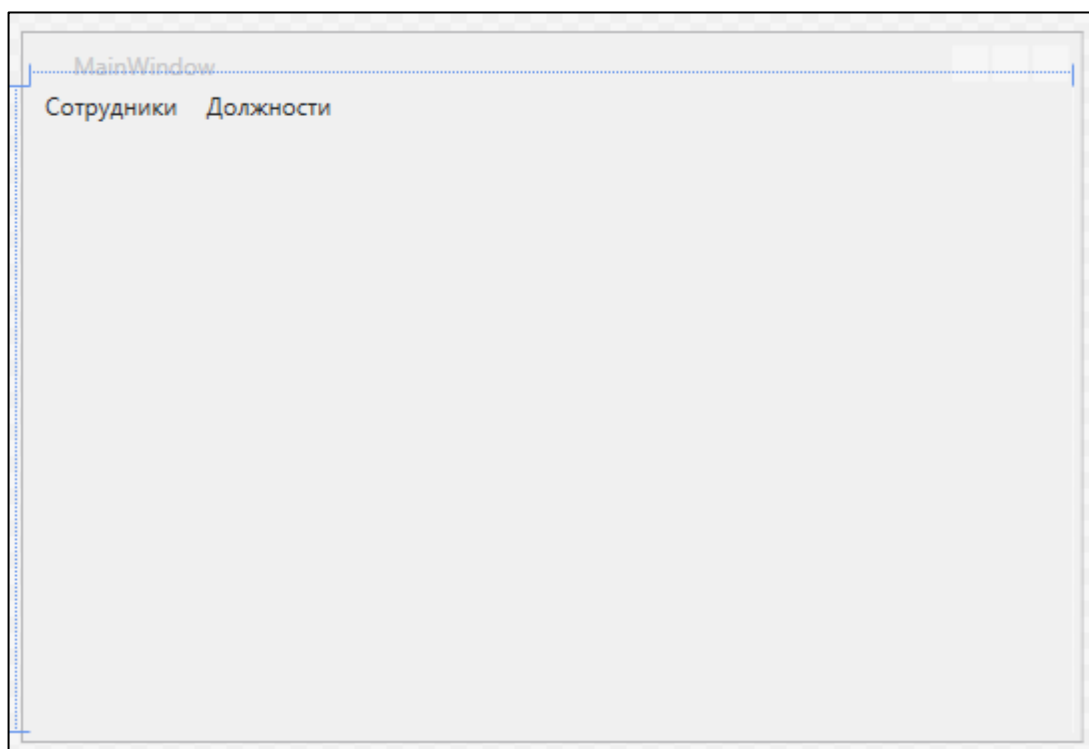


Рисунок 1.9 – главная форма с меню

XAML-разметка главной формы

```
<Window x:Class="WpfApplDemo2018.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:WpfApplDemo2018"
  mc:Ignorable="d"
  Title="MainWindow" Height="350" Width="525">
  <Grid>
    <Menu>
      <MenuItem x:Name="Employee" Header="Сотрудники" Click="Employee_OnClick"></MenuItem>
      <MenuItem x:Name="Role" Header="Должности" Click="Role_OnClick"></MenuItem>
    </Menu>
  </Grid>
</Window>
```

Для вызова окон для сотрудников и должностей добавим к код главного окна обработчики событий нажатия на соответствующих пунктах меню.

C# код главного окна

```
using System.Windows;
using WpfApplDemo2018.View;
namespace WpfApplDemo2018
{
  /// <summary>
  /// Логика взаимодействия для MainWindow.xaml
  /// </summary>
  public partial class MainWindow : Window
  {
    public MainWindow()
    {
      InitializeComponent();
    }
    private void Employee_OnClick(object sender, RoutedEventArgs e)
    {
      WindowEmployee wEmployee = new WindowEmployee();
      wEmployee.Show();
    }
    private void Role_OnClick(object sender, RoutedEventArgs e)
    {
      WindowRole wRole = new WindowRole();
      wRole.Show();
    }
  }
}
```

Тестирование результатов выполнения лабораторной работы

При сборке и выполнении приложения выводится главное окно (рисунок 1.10).

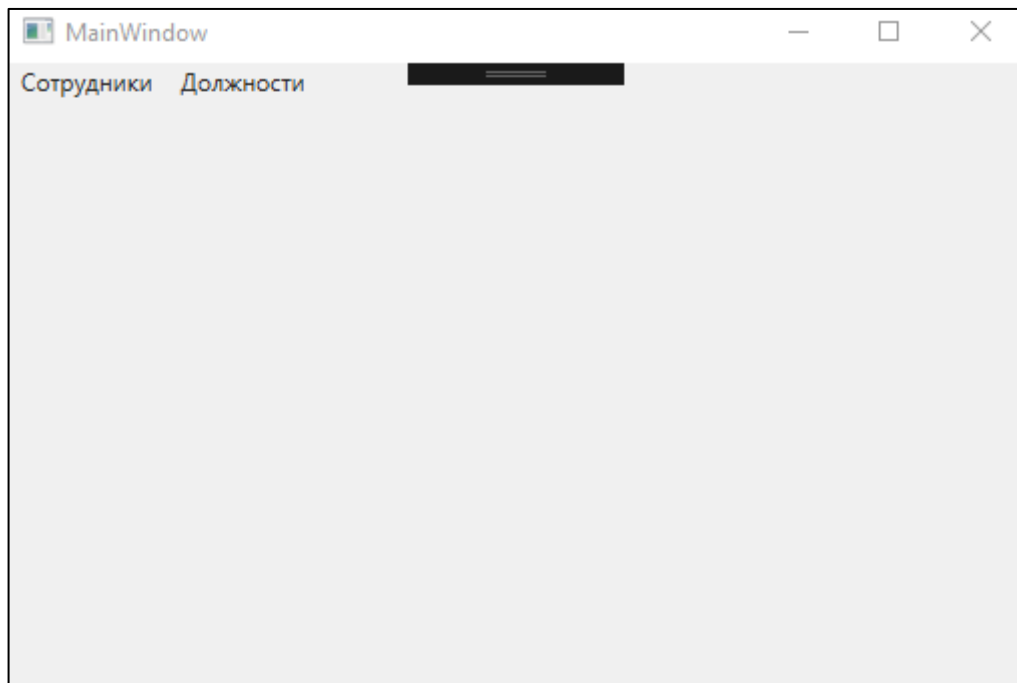


Рисунок 1.10 – главная форма

При нажатии на пункт меню «Сотрудники» выводится шаблон формы Сотрудники (рисунок 1.11).

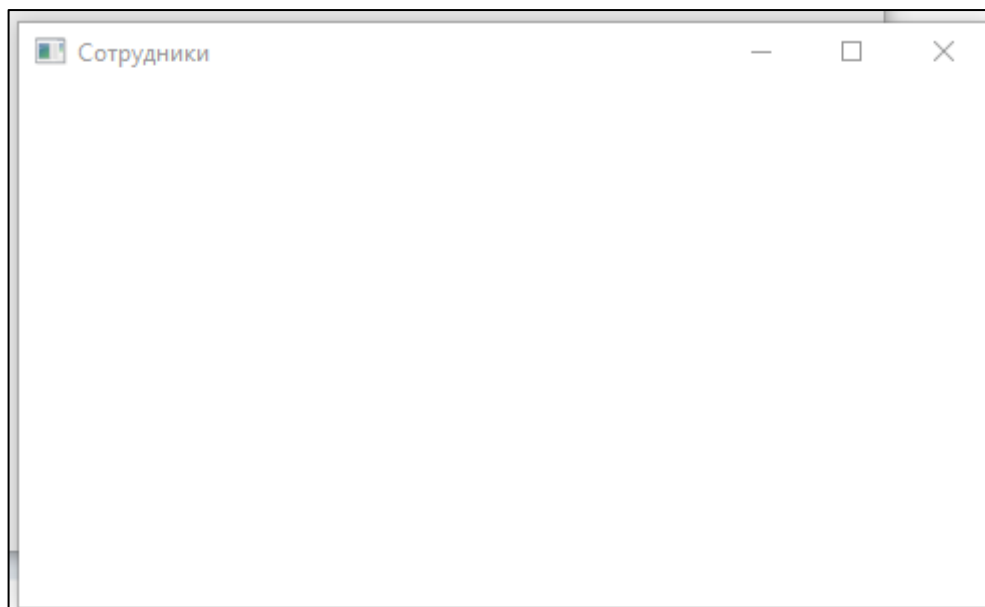


Рисунок 1.11 –форма Сотрудники

При нажатии на пункт меню «Должности» выводится шаблон формы Должности (рисунок 1.12).

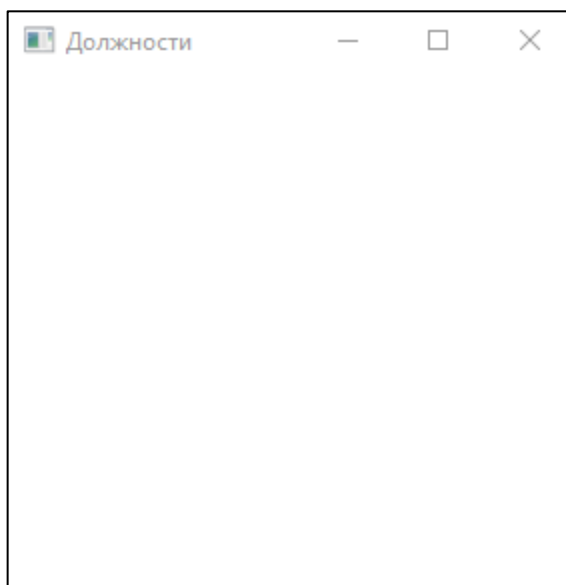


Рисунок 1.12 –форма Должности
Индивидуальные задания студент получает у преподавателя.

ЛАБОРАТОРНАЯ РАБОТА 2. Разработка представления данных по моделям

Цель работы: Создание представления моделей и отображение данных

Разработка представления для списка сотрудников

В лабораторной работе 1 был создан шаблон для экранной формы сотрудников (рисунок 2.1).

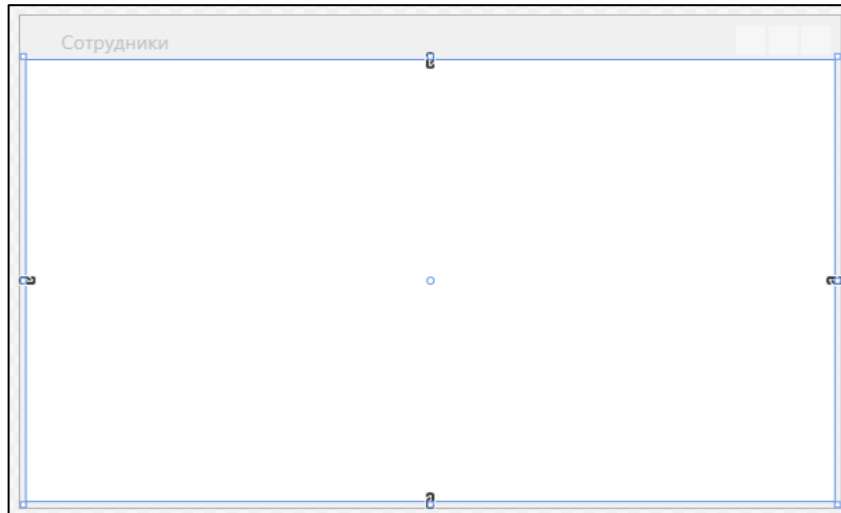


Рисунок 2.1 – Шаблон экранной формы для сотрудников

На данной форме добавим элементы контроля для представления информации по сотрудникам (рисунок 2.2).

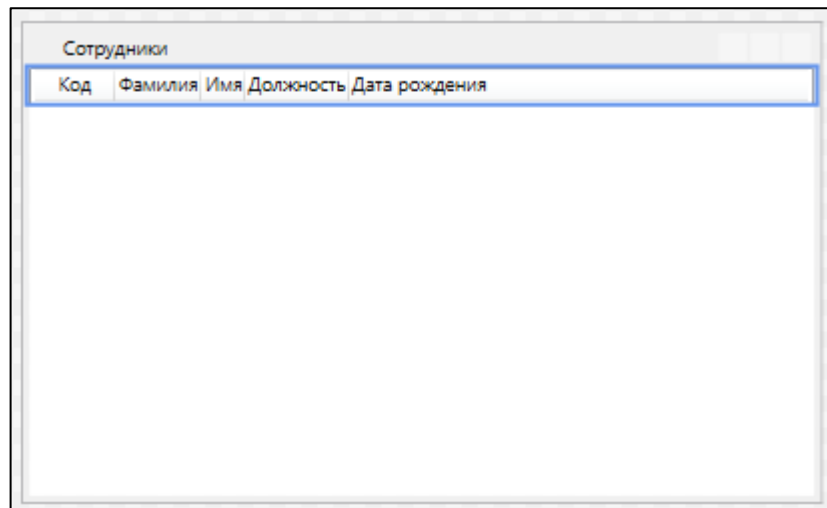


Рисунок 2.2 –Экранная форма Сотрудники

XAML-код окна WindowsEmployee

```
<Window x:Class="WpfApplDemo2018.View.WindowEmployee"
  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:WpfApplDemo2018.View"
  mc:Ignorable="d"
  Title="Сотрудники" Height="300" Width="500">
  <StackPanel>
    <ListView x:Name="lvEmployee">
      <ListView.View>
        <GridView>
          <GridView.Columns>
            <GridViewColumn Header="Код" Width="50" />
            <GridViewColumn Header="Фамилия" />
            <GridViewColumn Header="Имя" />
            <GridViewColumn Header="Должность" />
            <GridViewColumn Header="Дата рождения" />
          </GridView.Columns>
        </GridView>
      </ListView.View>
    </ListView>
  </StackPanel>
</Window>
```

Для отображения списка сотрудников будем использовать класс **ListView**, который предназначен для отображения данных спискового типа, состоящих из множества столбцов. Для класса **ListView** используется свойство **View (ListView.View)**, которое обеспечивает стили и форматирование данных. Для создания в отображении списка столбцов будем использовать класс **GridView**, который предоставляет списковое представление со множеством столбцов. Столбцы определяются в коллекции **GridView.Columns** путем задания объектов **GridViewColumn**. На данном этапе проектирования приложения для объектов **GridViewColumn** зададим только заголовок столбца **Header**.

Для отображения данных по сотрудникам необходимо в коде класса добавить следующий код, формирующий эти данные.

C# код класса WindowEmployee

```
using System.Windows;
using WpfApplDemo2018.ViewModel;
namespace WpfApplDemo2018.View
{
  /// <summary>
  /// Логика взаимодействия для WindowEmployee.xaml
  /// </summary>
  public partial class WindowEmployee : Window
```

```

{
    public WindowEmployee()
    {
        InitializeComponent();
        PersonViewModel vmPerson = new PersonViewModel();
        lvEmployee.ItemsSource = vmPerson.ListPerson;
    }
}

```

Создаем экземпляр класса **PersonViewModel** для формирования в программе данных по сотрудникам.

```
PersonViewModel vmPerson = new PersonViewModel();
```

Для отображения во **ListView** данных присваивает свойству **ItemsSource** коллекции с данными по сотрудникам.

```
lvEmployee.ItemsSource = vmPerson.ListPerson;
```

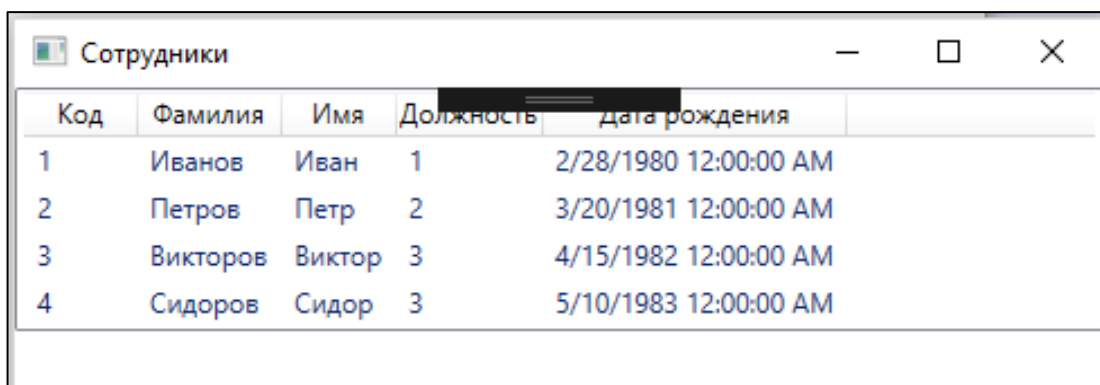
В XAML-коде для свойств **GridViewColumn** добавляем привязку к данным по сотрудникам. Привязка производится через свойство **DisplayMemberBinding**, которому через расширенную разметку присваивается класс **Binding** и требуемое свойство класса **Employee**, например **Id**.

```

<GridViewColumn Header="Код" Width="50"
    DisplayMemberBinding="{Binding Id}" />
<GridViewColumn Header="Фамилия"
    DisplayMemberBinding="{Binding LastName}" />
<GridViewColumn Header="Имя"
    DisplayMemberBinding="{Binding FirstName}" />
<GridViewColumn Header="Должность"
    DisplayMemberBinding="{Binding RoleId}" />
<GridViewColumn Header="Дата рождения"
    DisplayMemberBinding="{Binding Birthday}" />

```

После сборки и запуска программы при вызове окна Сотрудники экранная форма будет иметь вид, представленный на рисунке 2.3.



Код	Фамилия	Имя	Должность	Дата рождения
1	Иванов	Иван	1	2/28/1980 12:00:00 AM
2	Петров	Петр	2	3/20/1981 12:00:00 AM
3	Викторов	Виктор	3	4/15/1982 12:00:00 AM
4	Сидоров	Сидор	3	5/10/1983 12:00:00 AM

Рисунок 2.3 – Представление данных по сотрудникам

Аналогичным образом можно создать представление для вывода данных по должностям сотрудников (рисунок 2.4).

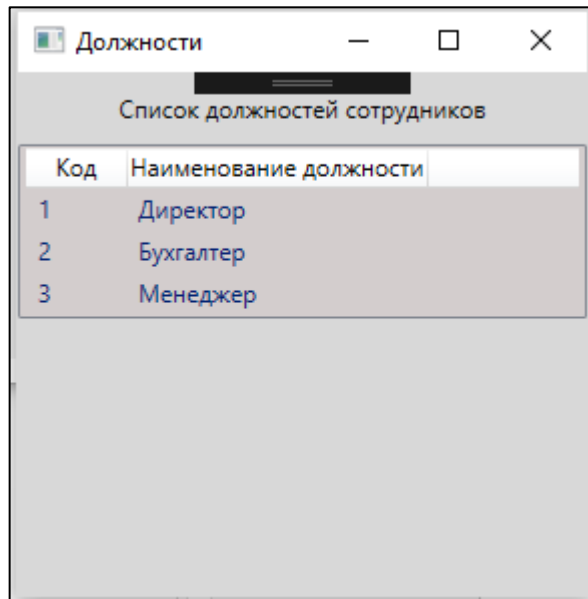


Рисунок 2.4 – Представление данных по должностям сотрудников
XAML-код окна WindowRole

```
<Window x:Class="WpfApplDemo2018.View.WindowRole"
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:WpfApplDemo2018.View"
mc:Ignorable="d"
Title="Должности" Height="300" Width="300">
<StackPanel Background="#FFD8D8D8">
<Label Margin="5" HorizontalAlignment="Center">Список должностей
сотрудников</Label>
<ListView x:Name="lvRole" Background="#FFD3CDCD">
<ListView.View>
<GridView>
<GridView.Columns>
<GridViewColumn Header="Код" Width="50"
DisplayMemberBinding="{Binding Id}"/>
<GridViewColumn Header="Наименование должности"
DisplayMemberBinding="{Binding NameRole}"/>
</GridView.Columns>
</GridView>
</ListView.View>
</ListView>
</StackPanel>
</Window>
```

При описании окна WindowRole добавлена закрашка фона панели StackPanel:

```
<StackPanel Background="#FFD8D8D8">
```

и элемента ListView:

```
<ListView x:Name="lvRole" Background="#FFD3CDCD">
```

Отображение должности в списке сотрудников

При выводе списка сотрудников (рисунок 2.3) вместо наименования должности выводится код должности. Для вывода наименования должности необходимо по коду найти соответствующую должность в классе должностей.

Для реализации этой возможности внесем изменения в приложение.

В проект добавим папку Helper (рисунок 2.5).

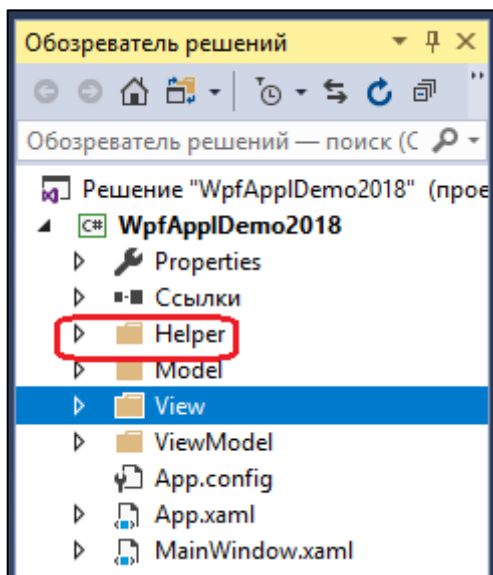


Рисунок 2.5 – Добавление папки Helper

В папке Helper создадим класс FindRole, который содержит предикат для поиска экземпляра класса Role по Id.

C# код класса FindRole

```
using System.Windows;
using WpfApplDemo2018.ViewModel;
namespace WpfApplDemo2018.Helper
{
    public class FindRole
    {
        int id;
        public FindRole(int id)
        {
            this.id = id;
        }
        public bool RolePredicate(Role role )
        {
            return role.Id == id;
        }
    }
}
```

Для отображения данных по сотруднику с наименованием должности создадим специальный класс PersonDPO, который отличается от класса Person тем, что вместо свойства RoleId имеет свойство Role строкового типа.

С# код класса PersonDPO

```

using System;
using System.Collections.Generic;
namespace WpfApplDemo2018.Model
{
    public class PersonDPO
    {
        public int Id { get; set; }
        public string Role { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime Birthday { get; set; }
        public PersonDPO() { }
        public PersonDPO(int id, string role, string firstName, string
lastName, DateTime birthday)
        {
            this.Id = id;
            this.Role = role;
            this.FirstName = firstName;
            this.LastName = lastName;
            this.Birthday = birthday;
        }
    }
}

```

Внесем изменения в класс WindowEmployee.cs для обеспечения возможности формирования новой обобщенной коллекции persons типа ObservableCollection<PersonDPO>.

С# код класса WindowEmployee.cs

```

using System.Windows;
using WpfApplDemo2018.Helper;
using WpfApplDemo2018.ViewModel;
using WpfApplDemo2018.Model;
using System.Collections.Generic;
using System;
using System.Collections.ObjectModel;
namespace WpfApplDemo2018.View
{
    /// <summary>
    /// Логика взаимодействия для WindowEmployee.xaml
    /// </summary>
    public partial class WindowEmployee : Window
    {
        public WindowEmployee()
        {
            InitializeComponent();
            PersonViewModel vmPerson = new PersonViewModel();
            RoleViewModel vmRole = new RoleViewModel();
            List<Role> roles = new List<Role>();
            foreach(Role r in vmRole.ListRole)
            {
                roles.Add(r);
            }
        }
    }
}

```

```

    }
    ObservableCollection<PersonDPO> persons = new ObservableCol-
lection<PersonDPO>();
    FindRole finder;
    foreach (var p in vmPerson.ListPerson)
    {
        finder = new FindRole(p.RoleId);
        Role rol = roles.Find(new Predicate<Role>(finder.RolePredi-
cate));
        persons.Add(new PersonDPO
        {
            Id = p.Id,
            Role = rol.NameRole,
            FirstName = p.FirstName,
            LastName = p.LastName,
            Birthday = p.Birthday
        });
    }
    lvEmployee.ItemsSource = persons;
}
}
}

```

Преобразуем коллекция классов Role типа ObservableCollection<Role> в обобщенную коллекцию roles типа List<Role>. Это необходимо сделать для того, чтобы можно было использовать метод Find() при поиске элемента в обобщенной коллекции.

```

RoleViewModel vmRole = new RoleViewModel();
List<Role> roles = new List<Role>();
foreach(Role r in vmRole.ListRole)
{
    roles.Add(r);
}

```

Далее создаем экземпляр обобщенной коллекции класса PersonDPO и объявляем экземпляр класса FindRole.

```

ObservableCollection<PersonDPO> persons = new ObservableCollec-
tion<PersonDPO>();
FindRole finder;

```

Организуем цикл по обобщенной коллекции ListPerson класса vmPerson. В цикле создаем экземпляр класса finder, передавая конструктору код должности сотрудник из класса Person - p.RoleId.

```

finder = new FindRole(p.RoleId);

```

Находим наименование должности с помощью метода Find(), который определен для обобщенных коллекций.

```

Role rol = roles.Find(new Predicate<Role>(finder.RolePredi-
cate));

```

Формируем коллекцию классов PersonDPO.

```

persons.Add(new PersonDPO
{
    Id = p.Id,

```

```

        Role = rol.NameRole,
        FirstName = p.FirstName,
        LastName = p.LastName,
        Birthday = p.Birthday
    });

```

Полный код цикла формирования обобщенной коллекции классов PersonDPO приведен ниже.

```

foreach (var p in vmPerson.ListPerson)
{
    finder = new FindRole(p.RoleId);
    Role rol = roles.Find(new Predicate<Role>(finder.RolePredicate));
    persons.Add(new PersonDPO
    {
        Id = p.Id,
        Role = rol.NameRole,
        FirstName = p.FirstName,
        LastName = p.LastName,
        Birthday = p.Birthday
    });
}

```

Последним шагом изменения кода класса является присвоение коллекции классов PersonDPO – persons источнику данных ListView/

```
lvEmployee.ItemsSource = persons;
```

Для правильного отображения должности сотрудника необходимо внести изменение в привязку данных окна WindowsEmployee, а также применим форматирование при выводе даты.

```

<GridViewColumn Header="Должность" Width="100"
    DisplayMemberBinding="{Binding Role}"/>
<GridViewColumn Header="Дата рождения"
    DisplayMemberBinding="{Binding Birthday,
        StringFormat={ }{0:dd\ } {0:MM\ } {0:yyyy} }"/>

```

После сборки и выполнения приложения окно со списком сотрудников будет выглядеть как представлено на рисунке 2.6.

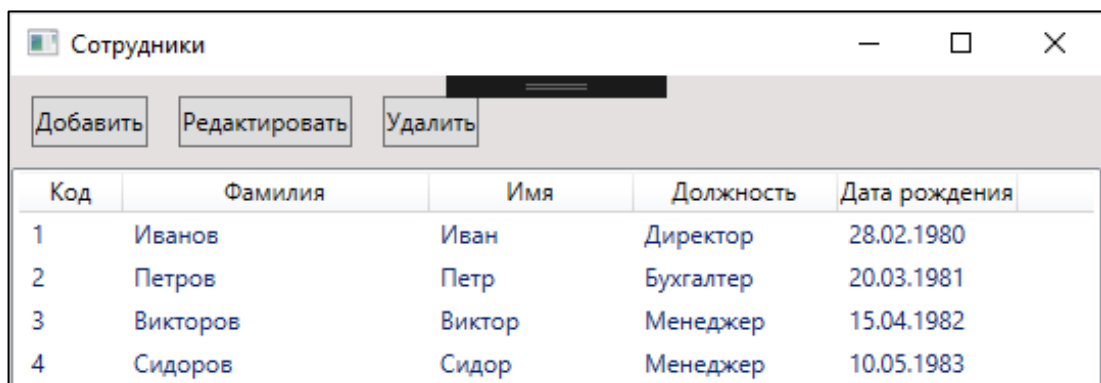


Рисунок 2.6 – Экранная форма Сотрудники

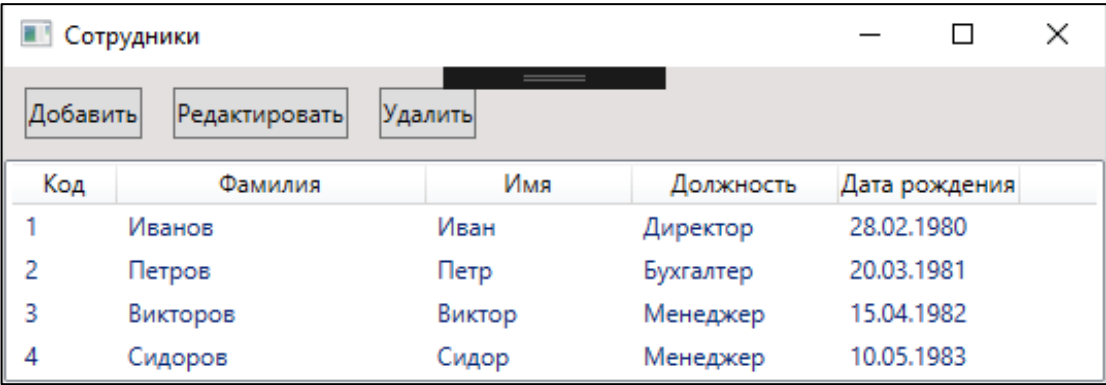
Задание на лабораторную работу

1. Разработать представление для классов модели данных в соответствии в варианте задания.
2. Протестировать отображение данных для каждого класса.
3. Внести изменения в приложение для отображения данных внешних классов, а не кодов.

ЛАБОРАТОРНАЯ РАБОТА 3. Создание, редактирование и удаление данных по должностям сотрудников

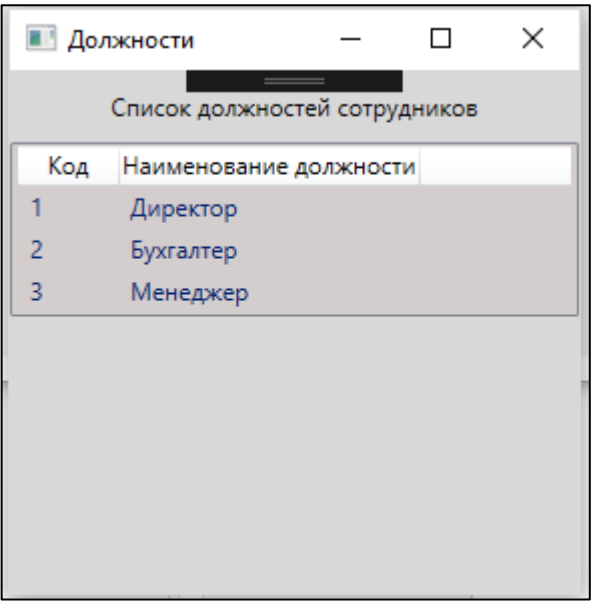
Цель работы: Манипулирование с данными предметной области

В лабораторной работе 4 была добавлена функциональность приложения в части отображения списка сотрудников и должностей (рисунок 3.1 и 3.2).



Код	Фамилия	Имя	Должность	Дата рождения
1	Иванов	Иван	Директор	28.02.1980
2	Петров	Петр	Бухгалтер	20.03.1981
3	Викторов	Виктор	Менеджер	15.04.1982
4	Сидоров	Сидор	Менеджер	10.05.1983

Рисунок 3.1 – Представление данных по сотрудникам



Код	Наименование должности
1	Директор
2	Бухгалтер
3	Менеджер

Рисунок 3.2 – Представление данных по должностям сотрудников

Для обеспечения функциональности обработки данных по должностям сотрудников добавим в окно *WindowsRole* кнопки *Добавить*, *Редактировать* и *Удалить* (рисунок 3.3).

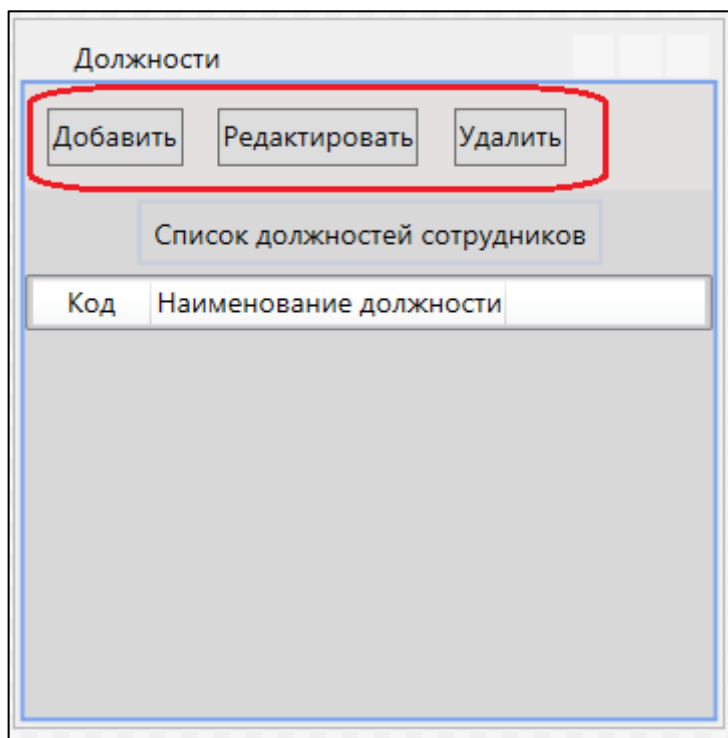


Рисунок 3.3 – Добавление кнопок на окно *Должности*

Кнопки расположим в контейнере *StackPanal* с горизонтальной ориентацией и в описании каждой кнопки добавим обработчик события *Click*.

Фрагмент XAML-разметки формы *WindowsRole*

```
<StackPanel Background="#FFD8D8D8">
    <StackPanel Orientation="Horizontal" Background="#FFE4E0E0">
        <Button x:Name="btnAdd" Margin="10,10,5,10" Content="Добавить"
            Height="25"
            Click="btnAdd_Click"/>
        <Button x:Name="btnEdit" Margin="10,10,5,10" Content="Редактировать"
            Height="25" Click="btnEdit_Click"/>
        <Button x:Name="btnDelete" Margin="10,10,5,10" Content="Удалить" Height="25"
            Click="btnDelete_Click"/>
    </StackPanel>
    <Label Margin="5" HorizontalAlignment="Center">
        Список должностей сотрудников</Label>
    <ListView x:Name="lvRole" Background="#FFD3CDCD">
        . . . .
        . . . .
        . . . .
    </ListView>
</StackPanel>
```

Добавление должности в список должностей сотрудников

Для формирования данных по новой должности создадим окно *WindowNewRole* (рисунок 3.4).

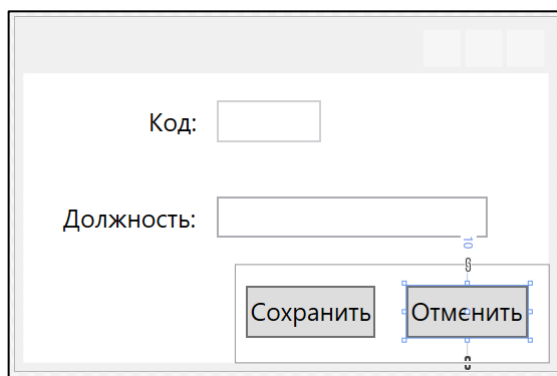


Рисунок 3.4 –Окно WindowNewRole
XAML-разметки формы WindowsNewRole

```
<Window x:Class="WpfApplDemo2018.View.WindowNewRole"
  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"
  mc:Ignorable="d"
  Height="170" Width="260">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="40*" />
      <RowDefinition Height="40*" />
      <RowDefinition Height="40*" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="40*" />
      <ColumnDefinition Width="75*" />
    </Grid.ColumnDefinitions>
    <TextBlock Grid.Row="0" Grid.Column="0" Text="Код:"
      HorizontalAlignment="Right" VerticalAlignment="Center"
      Margin="5" />
    <TextBlock Grid.Row="1" Grid.Column="0" Text="Должность:"
      HorizontalAlignment="Right" VerticalAlignment="Center"
      Margin="5" />
    <TextBox x:Name="TbId" Grid.Row="0" Grid.Column="1"
      Height="20" Width="50"
      HorizontalAlignment="Left" VerticalAlignment="Center" Margin="5"
      Text="{Binding Id}" IsEnabled="False" />
    <TextBox x:Name="TbRole" Grid.Row="1" Grid.Column="1"
      Height="20" Width="130"
      HorizontalAlignment="Left" VerticalAlignment="Center" Margin="5"
      Text="{Binding NameRole}" />
    <StackPanel Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="2"
      Orientation="Horizontal" HorizontalAlignment="Right">
      <Button x:Name="BtSave" Content="Сохранить" Height="25"
        HorizontalAlignment="Right" VerticalAlignment="Top" />
    </StackPanel>
  </Grid>
</Window>
```

```

        Margin="5,10,10,5" IsDefault="True" Click="BtSave_Click"/>
        <Button x:Name="BtCansel" Content="Отменить" Height="25"
            HorizontalAlignment="Right" VerticalAlignment="Top"
            Margin="5,10,10,5" IsCancel="True"/>
    </StackPanel>
</Grid>
</Window>

```

Код должности должен формироваться программно, поэтому для элемента *TbId* установлено свойство *IsEnabled="False"*, которое запрещает его редактирование. Кнопка *BtSave* (Сохранить) определяется как кнопка по умолчанию *IsDefault="True"*, кнопка *BtCansel* (Отменить), как кнопка отмены редактирования *IsCancel="True"*.

При создании новой должности её код должен вычисляться как максимальный код должности в имеющихся данных, т.е. в списке должностей, увеличенный на единицу. Для определения максимального значения кода в списке должностей добавим в класс *RoleViewModel* метод *MaxId()*.

Листинг метода *MaxId*

```

/// <summary>
/// Нахождение максимального Id
/// </summary>
/// <returns></returns>
public int MaxId()
{
    int max = 0;
    foreach (var r in this.ListRole)
    {
        if (max < r.Id)
        {
            max = r.Id;
        };
    }
    return max;
}

```

В коде класса *MainWinwow* добавим свойство *IdRole*.

```
public static int IdRole { get; set; }
```

При добавлении новой должности сотрудника создадим для кнопки *Добавить* окна *WindowsRole* код обработчика *btnAdd_Click*.

```

/// <summary>
/// Добавление новых данных по должности
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnAdd_Click(object sender, RoutedEventArgs e)
{
    WindowNewRole wnRole = new WindowNewRole
    {
        Title = "Новая должность",
        Owner = this
    };
}

```



```
// формирование кода новой должности
int maxIdRole = vmRole.MaxId() + 1;
Role role = new Role
{
    Id = maxIdRole
};
wnRole.DataContext = role;
if (wnRole.ShowDialog() == true)
{
    vmRole.ListRole.Add(role);
}
}
```

Создаем экземпляр окна WindowNewRole, задав свойству Title значение Новая должность, а свойству Owner (владелец) значение this, что определяет владельца окна WindowNewRole как WindowRole.

```
WindowNewRole wnRole = new WindowNewRole
{
    Title = "Новая должность",
    Owner = this
};
```

Определяем код для новой должности.

```
int maxIdRole = vmRole.MaxId() + 1;
```

Создаем экземпляр класса Role и задаем для свойства Id полученное значение кода должности.

```
int maxIdRole = vmRole.MaxId() + 1;
Role role = new Role
{
    Id = maxIdRole
};
```

Устанавливаем контекст для экземпляра окна wnRole.

```
wnRole.DataContext = role;
```

Открываем диалоговое окно wnRole для формирования данных по новой должности.

```
if (wnRole.ShowDialog() == true)
{
    vmRole.ListRole.Add(role);
}
```

Для добавление новой должности необходимо заполнить поле Должность и нажать кнопку Сохранить (рисунок 3.5).

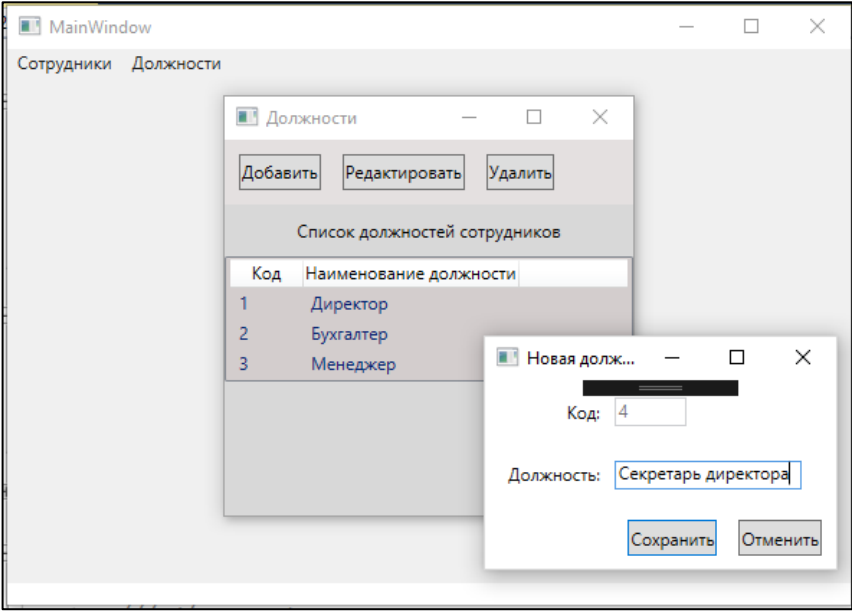


Рисунок 3.5 – Формирование новой должности

В обработчике кнопки Сохранить окна *WindowNewRole* свойству *DialogResult* присваивается значение *true* (значение по умолчанию *false*).

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

Если при закрытии окна *WindowNewRole* возвращаемое значение свойства *DialogResult = true*, то сформированный экземпляр класса *Role* добавляется в коллекцию классов *ListRole*.

```
vmRole.ListRole.Add(role);
```

Результат добавления новой должности приведен на рисунке 3.6

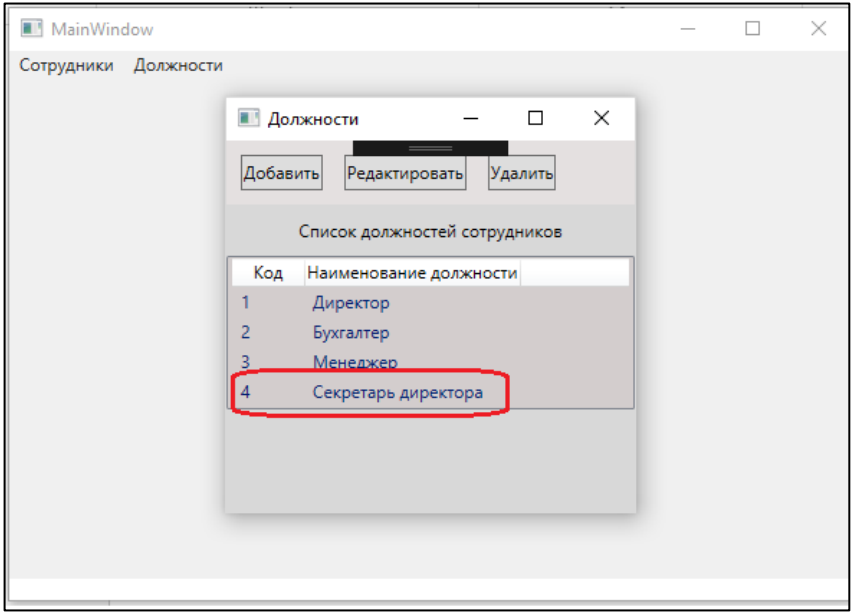


Рисунок 3.6 – Добавление новая должность

Если при формировании данных по новой должности в окне *WindowNewRole* нажать кнопку *Отменить*, то список должностей не изменится.

Редактирование данных по должности

При редактировании должности сотрудника создадим для кнопки *Редактировать* окна *WindowsRole* код обработчика *btnEdit_Click*.

```
private void btnEdit_Click(object sender, RoutedEventArgs e)
{
    WindowNewRole wnRole = new WindowNewRole
    {
        Title = "Редактирование должности",
        Owner = this
    };
    Role role = lvRole.SelectedItem as Role;
    if (role != null)
    {
        Role tempRole = role.ShallowCopy();
        wnRole.DataContext = tempRole;
        if (wnRole.ShowDialog() == true)
        {
            // сохранение данных
            role.NameRole = tempRole.NameRole;
            lvRole.ItemsSource = null;
            lvRole.ItemsSource = vmRole.ListRole;
        }
    }
    else
    {
        MessageBox.Show("Необходимо выбрать должность для редактирования",
            "Предупреждение", MessageBoxButton.OK, MessageBoxImage.Warning);
    }
}
```

При редактировании данных по должности создаем экземпляр окна *WindowNewRole*.

```
WindowNewRole wnRole = new WindowNewRole
{
    Title = "Редактирование должности",
    Owner = this
};
```

Для редактирования данных по должности формируем экземпляр *role* класса *Role* из выделенного элемента списка должностей *lvRole*.

```
Role role = lvRole.SelectedItem as Role;
```

Проверяем наличие выделенного элемента в списке *lvRole*.

```
if (role != null)
```

Если в списке *lvRole* выделенный элемент отсутствует, то выводим сообщение (рисунок 3.7).

```

MessageBox.Show("Необходимо выбрать должность для редактирования",
    "Предупреждение", MessageBoxButtons.OK, MessageBoxIcon.Warning);

```

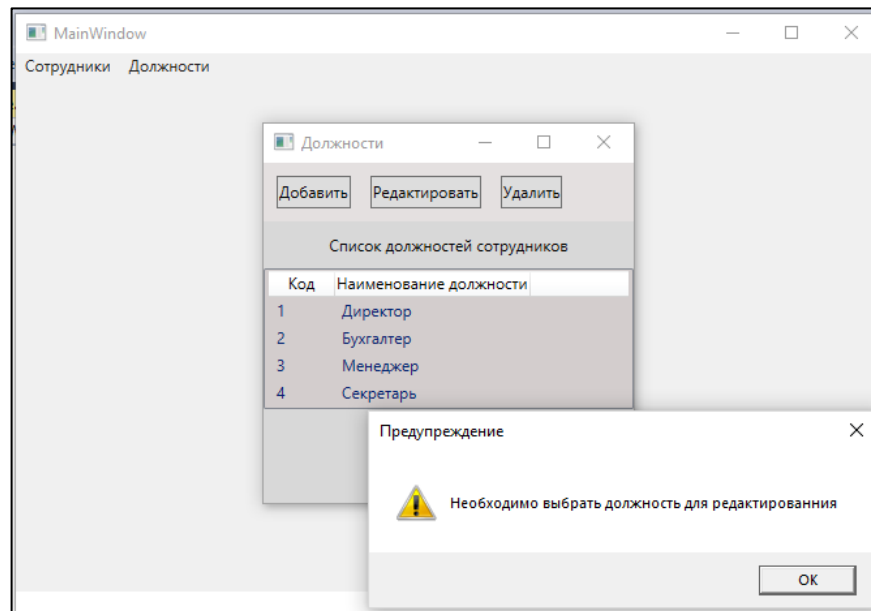


Рисунок 3.7 – Предупреждение о необходимости выбора строки списка

Если выделенный элемент в списке *lvRole* имеется, то для редактирования данных по должности необходимо создать копию экземпляра *role* класса *Role*. Метод создания поверхностной копии добавим в класс *Role*.

```

public Role ShallowCopy()
{
    return (Role)this.MemberwiseClone();
}

```

Применим метод *ShallowCopy()* для создания копии *tempRole* экземпляра *role* класса *Role*.

```

Role tempRole = role.ShallowCopy();

```

Установим экземпляр *tempRole* класса *Role* в качестве контекста окна *wnRole* редактирования данных по должности

```

wnRole.DataContext = tempRole;

```

В процессе редактирования открывается окно *Редактирования данных* по должностям сотрудников. Процесс редактирования наименования должности «Секретарь директора» на «Секретарь» приведен на рисунке 3.8.

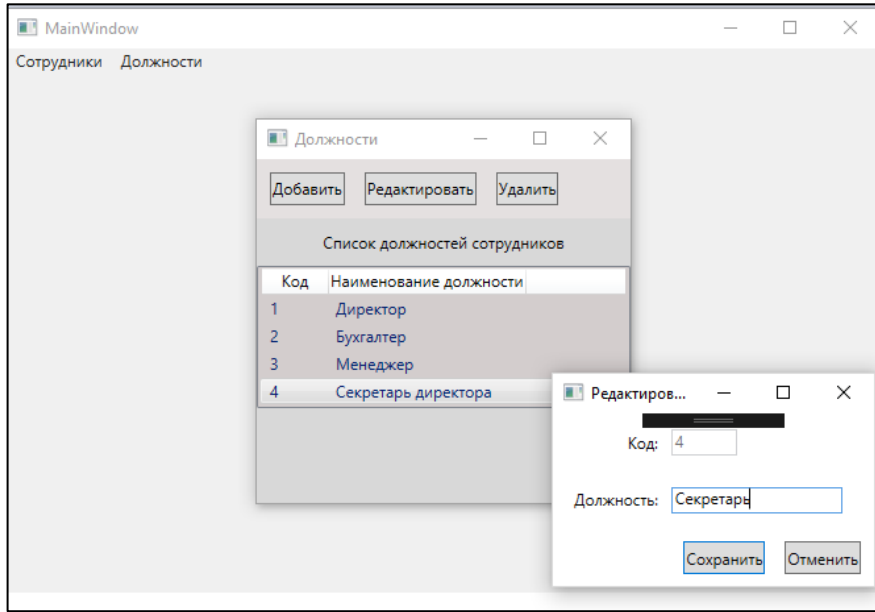


Рисунок 3.8 – Редактирование наименования должности

Если результаты редактирования подтверждается, т.е. нажимается кнопка *Сохранить*, то в программе возвращается значение true свойства *DialogResult* и выполняется процесс сохранения редактируемых данных и обновление привязки для списка должностей.

```
if (wnRole.ShowDialog() == true)
{
    // сохранение данных
    role.NameRole = tempRole.NameRole;
    lvRole.ItemsSource = null;
    lvRole.ItemsSource = vmRole.ListRole;
}
```

Результат редактирования приведен на рисунке 3.9.

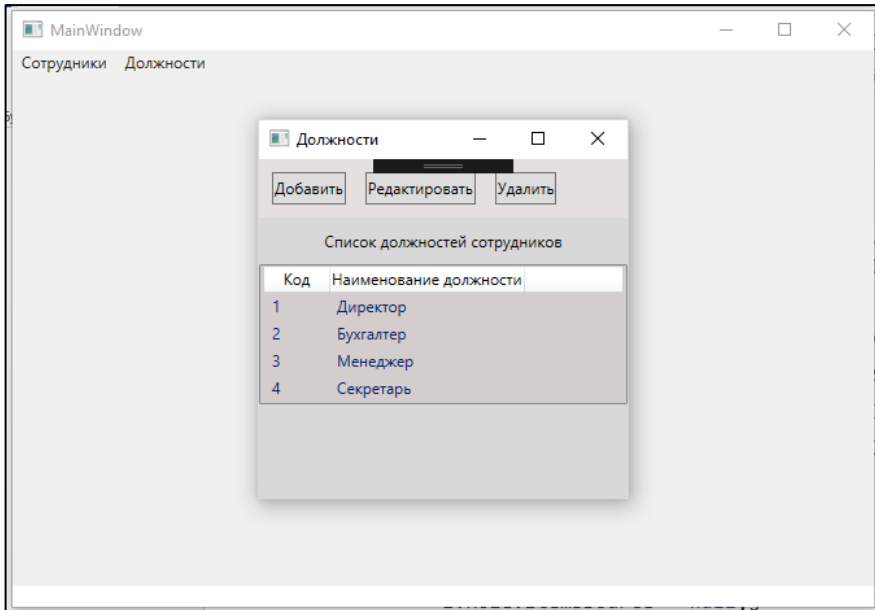


Рисунок 3.9 – Результат редактирования наименования должности

Удаление должности из списка должностей сотрудников

При удалении должности сотрудника создадим для кнопки *Удалить* окна *WindowsRole* код обработчика *btnEdit_Click*.

```
private void btnDelete_Click(object sender, RoutedEventArgs e)
{
    Role role = (Role)lvRole.SelectedItem;
    if (role != null)
    {
        MessageBoxResult result = MessageBox.Show("Удалить данные по должности: " +
            role.NameRole, "Предупреждение", MessageBoxButton.OKCancel,
            MessageBoxImage.Warning);
        if (result == MessageBoxResult.OK)
        {
            vmRole.ListRole.Remove(role);
        }
    }
    else
    {
        MessageBox.Show("Необходимо выбрать должность для удаления",
            "Предупреждение", MessageBoxButton.OK, MessageBoxImage.Warning);
    }
}
```

При удалении должности формируем экземпляр *role* класса *Role* из выделенного элемента списка должностей *lvRole*.

```
Role role = (Role)lvRole.SelectedItem;
```

Если выбор должности для удамеления произведен, то выводится предупреждение (рисунок 3.10).

```
MessageBoxResult result = MessageBox.Show("Удалить данные по должности: " +
    role.NameRole, "Предупреждение", MessageBoxButton.OKCancel,
    MessageBoxImage.Warning);
```

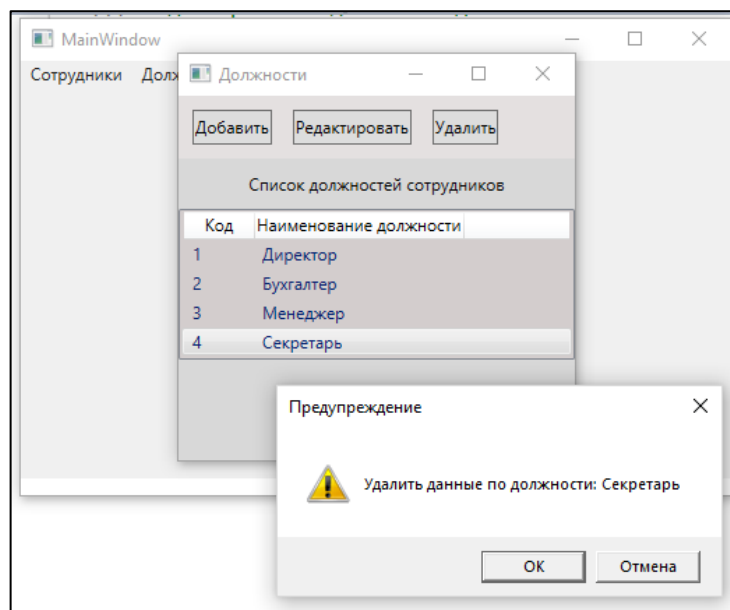


Рисунок 3.10 – Предупреждение об удалении должности

Если удаление подтверждается, то выбранный экземпляр должности удаляется из списка должностей `ListRole`.

```
if (result == DialogResult.OK)
{
    vmRole.ListRole.Remove(role);
}
```

Задание на лабораторную работу

1. Добавить в приложение функциональность для добавления, редактирования и удаление информации по классам модели данных в соответствии в варианте задания.
2. Протестировать добавленную функциональность.

ЛАБОРАТОРНАЯ РАБОТА 4. Создание, редактирование и удаление данных по сотрудникам

Цель работы: Манипулирование с данными классов предметной области

Для обеспечения функциональности обработки данных по сотрудникам добавим в окно *WindowEmployee* кнопки *Добавить*, *Редактировать* и *Удалить* (рисунок 4.1).

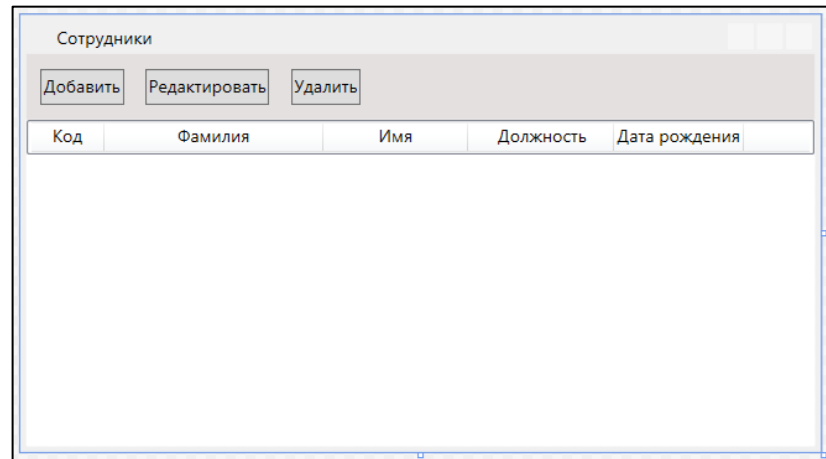


Рисунок 4.1 – Добавление кнопок на окно *Сотрудники*

Добавление сотрудника в список сотрудников

Для формирования данных по новому сотруднику создадим окно *WindowNewEmployee* (рисунок 4.2).

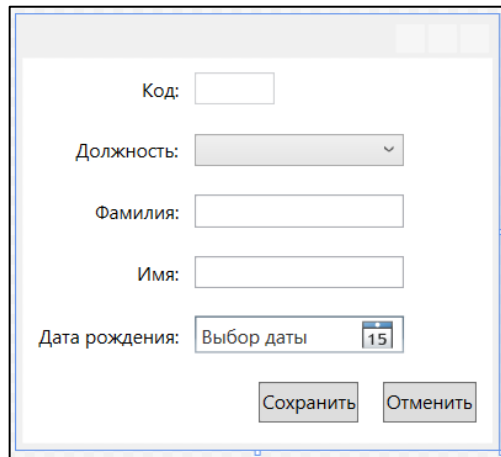


Рисунок 4.2 –Окно *WindowNewEmployee*

XAML-разметка формы WindowNewEmployee

```
<Window x:Class="WpfApplDemo2018.View.WindowNewEmployee"
  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"
  mc:Ignorable="d"
  Height="270" Width="300">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="40*" />
      <RowDefinition Height="40*" />
      <RowDefinition Height="40*" />
      <RowDefinition Height="40*" />
      <RowDefinition Height="40*" />
      <RowDefinition Height="50*" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="40*" />
      <ColumnDefinition Width="75*" />
    </Grid.ColumnDefinitions>
    <TextBlock Grid.Row="0" Grid.Column="0"
      Text="Код:" HorizontalAlignment="Right"
      VerticalAlignment="Center" Margin="5"/>
    <TextBlock Grid.Row="1" Grid.Column="0"
      Text="Должность:" HorizontalAlignment="Right"
      VerticalAlignment="Center" Margin="5"/>
    <TextBlock Grid.Row="2" Grid.Column="0"
      Text="Фамилия:" HorizontalAlignment="Right"
      VerticalAlignment="Center" Margin="5"/>
    <TextBlock Grid.Row="3" Grid.Column="0"
      Text="Имя:" HorizontalAlignment="Right"
      VerticalAlignment="Center" Margin="5"/>
    <TextBlock Grid.Row="4" Grid.Column="0"
      Text="Дата рождения:" HorizontalAlignment="Right"
      VerticalAlignment="Center" Margin="5"/>
    <TextBox x:Name="TbId" Grid.Row="0" Grid.Column="1"
      Text="{Binding Id}" Height="20" Width="50"
      HorizontalAlignment="Left" VerticalAlignment="Center"
      Margin="5" IsEnabled="False"/>
    <TextBox x:Name="TbLastName" Grid.Row="2" Grid.Column="1"
      Text="{Binding LastName}"
      Height="20" Width="130"
      HorizontalAlignment="Left" VerticalAlignment="Center" Margin="5"/>
    <TextBox x:Name="TbFirstName" Grid.Row="3" Grid.Column="1"
      Text="{Binding FirstName}"
      Height="20" Width="130"
      HorizontalAlignment="Left" VerticalAlignment="Center" Margin="5"/>
```

```

<ComboBox x:Name="CbRole" Grid.Row="1" Grid.Column="1"
Height="20" Width="130"
    HorizontalAlignment="Left" VerticalAlignment="Center" Mar-
gin="5"
    DisplayMemberPath="NameRole"/>
<DatePicker x:Name="ClBirthday" Grid.Row="4" Grid.Column="1"
    SelectedDate="{Binding Birthday, Margin="5,8,0,7"
    Mode=TwoWay,UpdateSourceTrigger=PropertyChanged}"
    HorizontalAlignment="Left" VerticalAlignment="Center"
Width="130"/>
<StackPanel Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="5"
    Orientation="Horizontal"
    HorizontalAlignment="Right">
    <Button x:Name="BtSave" Content="Сохранить" Height="25"
        HorizontalAlignment="Right" VerticalAlignment="Top"
        Margin="5,10,10,5"
        Click="BtSave_Click"/>
    <Button x:Name="BtCancel" Content="Отменить" Height="25"
        HorizontalAlignment="Right" VerticalAlignment="Top"
        Margin="5,10,10,5" IsCancel="True"/>
</StackPanel>
</Grid>
</Window>

```

При создании нового сотрудника его код должен вычисляться как максимальный код сотрудника в имеющихся данных, т.е. в списке сотрудников, увеличенный на единицу. Для определения максимального значения кода в списке должностей добавим в класс *PersonViewModel* метод *MaxId()*.

Листинг метода MaxId

```

public int MaxId()
{
    int max = 0;
    foreach (var r in this.ListPerson)
    {
        if (max < r.Id)
        {
            max = r.Id;
        };
    }
    return max;
}

```

В коде класса *MainWinwow* добавим свойство *IdEmployee*.

```

public static int IdEmployee { get; set; }

```

При выполнении операций обработки данных по сотрудникам неоднократно приходится формировать экземпляр класса *Person* из экземпляра класса *PersonDPO*, заменяя наименование должности на её код, и наоборот экземпляр класса *PersonDPO* из экземпляра класса *Person*, заменяя код должности на её наименование. Для выполнения такого преобразования создадим

В

классе

Person метод *CopyFromPersonDPO()*, а в классе *PersonDPO* метод *CopyFromPerson()*

```
public Person CopyFromPersonDPO(PersonDPO p)
{
    RoleViewModel vmRole = new RoleViewModel();
    int roleId = 0;
    foreach (var r in vmRole.ListRole)
    {
        if (r.NameRole == p.Role)
        {
            roleId = r.Id;
            break;
        }
    }
    if (roleId != 0)
    {
        this.Id = p.Id;
        this.RoleId = roleId;
        this.FirstName = p.FirstName;
        this.LastName = p.LastName;
        this.Birthday = p.Birthday;
    }
    return this;
}

public PersonDPO CopyFromPerson(Person person)
{
    PersonDPO perDPO = new PersonDPO();
    RoleViewModel vmRole = new RoleViewModel();
    string role = string.Empty;
    foreach (var r in vmRole.ListRole)
    {
        if (r.Id == person.RoleId)
        {
            role = r.NameRole;
            break;
        }
    }
    if (role != string.Empty)
    {
        perDPO.Id = person.Id;
        perDPO.Role = role;
        perDPO.FirstName = person.FirstName;
        perDPO.LastName = person.LastName;
        perDPO.Birthday = person.Birthday;
    }
    return perDPO;
}
```

Метод *CopyFromPersonDPO()* формирует класс *Person*, заменяя наименование должности *Role* на её код *roleId*, используя поиск в списке *ListRole*.

```
foreach (var r in vmRole.ListRole)
{
    if (r.NameRole == p.Role)
    {
        roleId = r.Id;
        break;
    }
}
```

Метод *CopyFromPerson()* формирует класс *PersonDPO*, заменяя код *roleId* на наименование должности *Role*.

```
foreach (var r in vmRole.ListRole)
{
    if (r.Id == person.Id)
    {
        role = r.NameRole;
        break;
    }
}
```

В класс *WindowEmployee* объявим поля *vmPerson*, *vmRole*, *personsDPO* и *roles*.

```
private PersonViewModel vmPerson = MainWindow.vmPerson;
private RoleViewModel vmRole;
private ObservableCollection<PersonDPO> personsDPO;
private List<Role> roles;
```

Конструктор класса *WindowEmployee* изменим следующим образом.

```
public WindowEmployee()
{
    InitializeComponent();
    vmPerson = new PersonViewModel();
    vmRole = new RoleViewModel();
    roles = vmRole.ListRole.ToList();
    // Формирование данных для отображения сотрудников с должностями
    // на базе коллекции класса ListPerson<Person>
    personsDPO = new ObservableCollection<PersonDPO>();
    foreach (var person in vmPerson.ListPerson)
    {
        PersonDPO p = new PersonDPO();
        p = p.CopyFromPerson(person);
        personsDPO.Add(p);
    }
    lvEmployee.ItemsSource = personsDPO;
}
```

После инициализации окна создадим экземпляры классов *vmPerson*, *vmRole*, *roles* и *personsDPO*.

```
vmPerson = new PersonViewModel();
vmRole = new RoleViewModel();
roles = vmRole.ListRole.ToList();
personsDPO = new ObservableCollection<PersonDPO>();
```

Далее сформируем коллекцию *personsDPO* классов *PersonDPO* для отображения в окне *WindowEmployee*.

```
foreach (var person in vmPerson.ListPerson)
{
    PersonDPO p = new PersonDPO();
    p = p.CopyFromPerson(person);
    personsDPO.Add(p);
}
```

Для списка *lvEmployee* зададим источник данных *ItemsSource*.

```
lvEmployee.ItemsSource = personsDPO;
```

При добавлении новой должности сотрудника необходимо создать для кнопки *Добавить* окна *WindowsRole* код обработчика *btnAdd_Click*.

```
private void btnAdd_Click(object sender, RoutedEventArgs e)
{
    WindowNewEmployee wnEmployee = new WindowNewEmployee
    {
        Title = "Новый сотрудник",
        Owner = this
    };
    // формирование кода нового сотрудника
    int maxIdPerson = vmPerson.MaxId() + 1;
    PersonDPO per = new PersonDPO
    {
        Id = maxIdPerson,
        Birthday = DateTime.Now
    };
    wnEmployee.DataContext = per;
    wnEmployee.CbRole.ItemsSource = roles;
    if (wnEmployee.ShowDialog() == true)
    {
        Role r = (Role)wnEmployee.CbRole.SelectedValue;
        per.Role = r.NameRole;
        personsDPO.Add(per);
        // добавление нового сотрудника в коллекцию ListPerson<Person>
        Person p = new Person();
        p = p.CopyFromPersonDPO(per);
        vmPerson.ListPerson.Add(p);
    }
}
```

Создаем экземпляр *wnEmployee* окна *WindowNewEmployee*.

```
WindowNewEmployee wnEmployee = new WindowNewEmployee
{
    Title = "Новый сотрудник",
    Owner = this
};
```

Определяем код *maxIdPerson* для нового сотрудника и создаем экземпляр *per* класса *PersonDPO*.

```

int maxIdPerson = vmPerson.MaxId() + 1;
PersonDPO per = new PersonDPO
{
    Id = maxIdPerson,
    Birthday = DateTime.Now
};

```

Задаем контекст данных для окна *wnEmployee* и источник данных для выпадающего списка *CbRole*.

```

wnEmployee.DataContext = per;
wnEmployee.CbRole.ItemsSource = roles;

```

На рисунке 4.3 представлена экранная форма формирования данных по новому сотруднику.

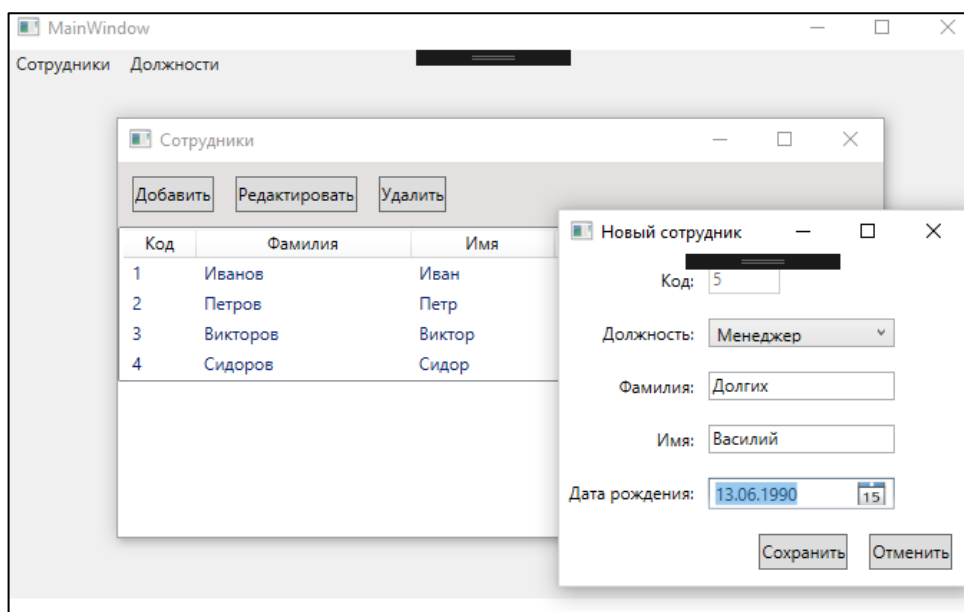


Рисунок 4.3 – Добавление нового сотрудника

При нажатии кнопки Сохранить (подтверждение ввода данных по новому сотруднику) осуществляется формирование данных и добавление в коллекции *personsDPO* и *vmPerson.ListPerson*.

```

if (wnEmployee.ShowDialog() == true)
{
    Role r = (Role)wnEmployee.CbRole.SelectedValue;
    per.Role = r.NameRole;
    personsDPO.Add(per);
    Person p = new Person();
    p = p.CopyFromPersonDPO(per);
    vmPerson.ListPerson.Add(p);
}

```

На рисунке 4.4 представлена экранная форма результата добавления данных по новому сотруднику.

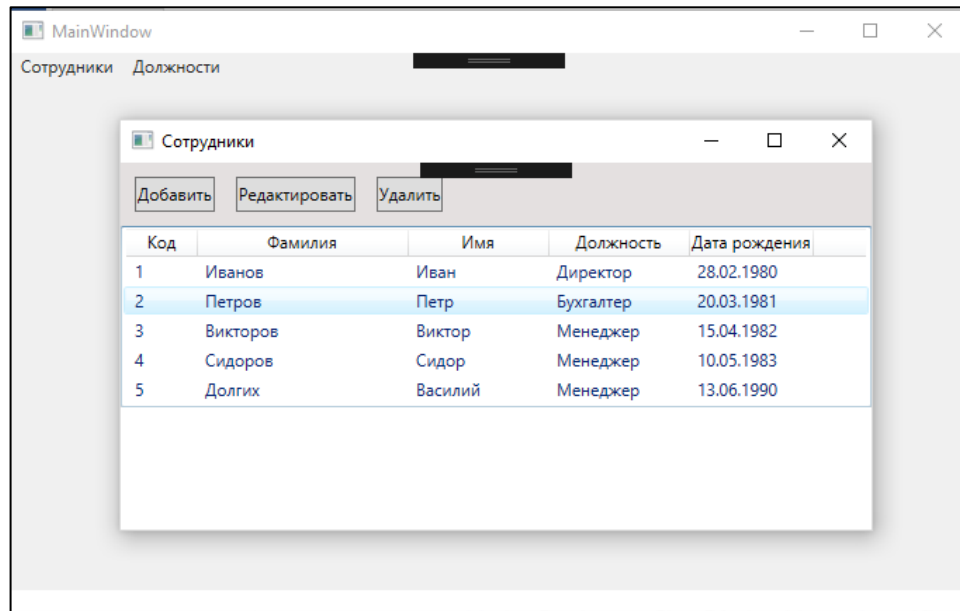


Рисунок 4.4 – Результат добавления нового сотрудника

Если при формировании данных по новому сотруднику в окне *WindowNewEmployee* нажать кнопку *Отменить*, то список сотрудников не изменится.

Редактирование данных по сотруднику

При редактировании данных по сотруднику создадим для кнопки *Редактировать* окна *WindowsEmployee* код обработчика *btnEdit_Click*.

```
private void btnEdit_Click(object sender, RoutedEventArgs e)
{
    WindowNewEmployee wnEmployee = new WindowNewEmployee
    {
        Title = "Редактирование данных",
        Owner = this
    };
    PersonDPO perDPO = (PersonDPO)lvEmployee.SelectedValue;
    PersonDPO tempPerDPO; // временный класс для редактирования
    if (perDPO != null)
    {
        tempPerDPO = perDPO.ShallowCopy();
        wnEmployee.DataContext = tempPerDPO;
        wnEmployee.CbRole.ItemsSource = roles;
        wnEmployee.CbRole.Text = tempPerDPO.Role;
        if (wnEmployee.ShowDialog() == true)
        {
            // перенос данных из временного класса в класс отображения
            данных
            Role r = (Role)wnEmployee.CbRole.SelectedValue;
            perDPO.Role = r.NameRole;
            perDPO.FirstName = tempPerDPO.FirstName;
            perDPO.LastName = tempPerDPO.LastName;
            perDPO.Birthday = tempPerDPO.Birthday;
            lvEmployee.ItemsSource = null;
            lvEmployee.ItemsSource = personsDPO;
        }
    }
}
```

```

        // перенос данных из класса отображения данных в класс Per-
son
        FindPerson finder = new FindPerson(perDPO.Id);
        List<Person> listPerson = vmPerson.ListPerson.ToList();
        Person p = listPerson.Find(new Predicate<Per-
son>(finder.PersonPredicate));
        p = p.CopyFromPersonDPO(perDPO);
    }
}
else
{
    MessageBox.Show("Необходимо выбрать сотрудника для редакти-
рования",
        "Предупреждение", MessageBoxButton.OK, MessageBoxI-
mage.Warning);
}
}

```

При редактировании данных по должности создаем экземпляр окна *Win-
dowNewEmployee*.

```

WindowNewEmployee wnEmployee = new WindowNewEmployee
{
    Title = "Редактирование данных",
    Owner = this
};

```

Для редактирования данных по сотруднику формируем экземпляр *perDPO* класса *PersonDPO* из выделенного элемента списка сотрудников *lvEmployee* и объявляем временный экземпляр *tempPerDPO* класса *PersonDPO*, который используется для редактирования.

```

PersonDPO perDPO = (PersonDPO)lvEmployee.SelectedValue;
PersonDPO tempPerDPO;

```

Проверяем наличие выделенного элемента в списке *lvEmployee*.

```

if (perDPO != null)

```

Если в списке *lvEmployee* выделенный элемент отсутствует, то выводим сообщение (рисунок 4.5).

```

MessageBox.Show("Необходимо выбрать сотрудника для редактирован-
ния",
    "Предупреждение", MessageBoxButton.OK, MessageBoxImage.Warn-
ing);

```

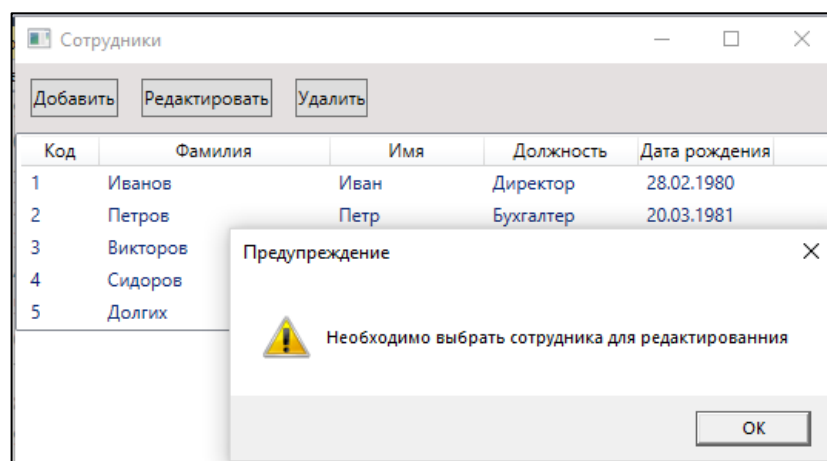



Рисунок 4.5 – Предупреждение о необходимости выбора строки списка

Если выделенный элемент в списке *lvEmployee* имеется, то для редактирования данных по сотруднику необходимо создать копию экземпляра *tempPerDPO* класса *PersonDPO*. Метод создания поверхностной копии добавим в класс *PersonDPO*.

```
public PersonDPO ShallowCopy()
{
    return (PersonDPO)this.MemberwiseClone();
}
```

Применим метод *ShallowCopy()* для создания копии *tempPerDPO* экземпляра *perDPO* класса *PersonDPO*.

```
tempPerDPO = perDPO.ShallowCopy();
```

Установим экземпляр *tempPerDPO* класса *PersonDPO* в качестве контекста окна *wnEmployee* редактирования данных по сотруднику и источник данных для выпадающего списка *CbRole*.

```
wnEmployee.DataContext = tempPerDPO;
wnEmployee.CbRole.ItemsSource = roles;
wnEmployee.CbRole.Text = tempPerDPO.Role;
```

В процессе редактирования открывается окно *Редактирования данных* по сотрудникам. Процесс редактирования данных по сотруднику изменим фамилию Долгих на Долгов и дату рождения с 13.06.1990 г. на 13.07.1990 г. Результат редактирования данных по сотруднику приведен на рисунке 4.6.

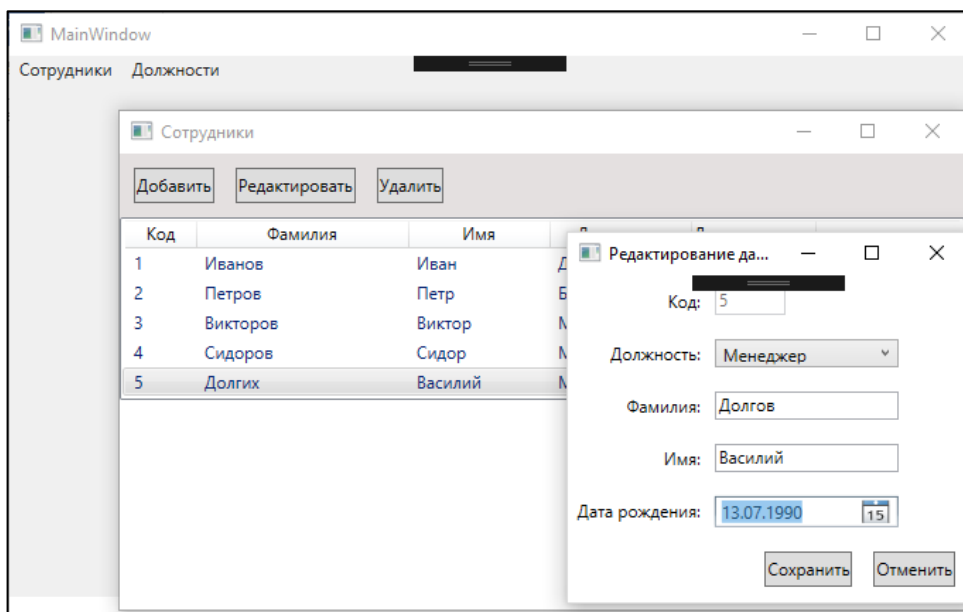


Рисунок 4.6 – Редактирование данных по сотруднику

Если результаты редактирования подтверждаются, т.е. нажимается кнопка *Сохранить*, то в программе возвращается значение true свойства *DialogResult* и выполняется процесс сохранения редактируемых данных и обновление привязки для списка сотрудников.

```
if (wnEmployee.ShowDialog() == true)
{
    // перенос данных из временного класса в класс отображения дан-
    ных
    Role r = (Role)wnEmployee.CbRole.SelectedValue;
    perDPO.Role = r.NameRole;
    perDPO.FirstName = tempPerDPO.FirstName;
    perDPO.LastName = tempPerDPO.LastName;
    perDPO.Birthday = tempPerDPO.Birthday;
    lvEmployee.ItemsSource = null;
    lvEmployee.ItemsSource = personsDPO;
    // перенос данных из класса отображения данных в класс Person
    FindPerson finder = new FindPerson(perDPO.Id);
    List<Person> listPerson = vmPerson.ListPerson.ToList();
    Person p = listPerson.Find(new Predicate<Person>(finder.Per-
    sonPredicate));
    p = p.CopyFromPersonDPO(perDPO);
}
```

Результат редактирования приведен на рисунке 4.7.

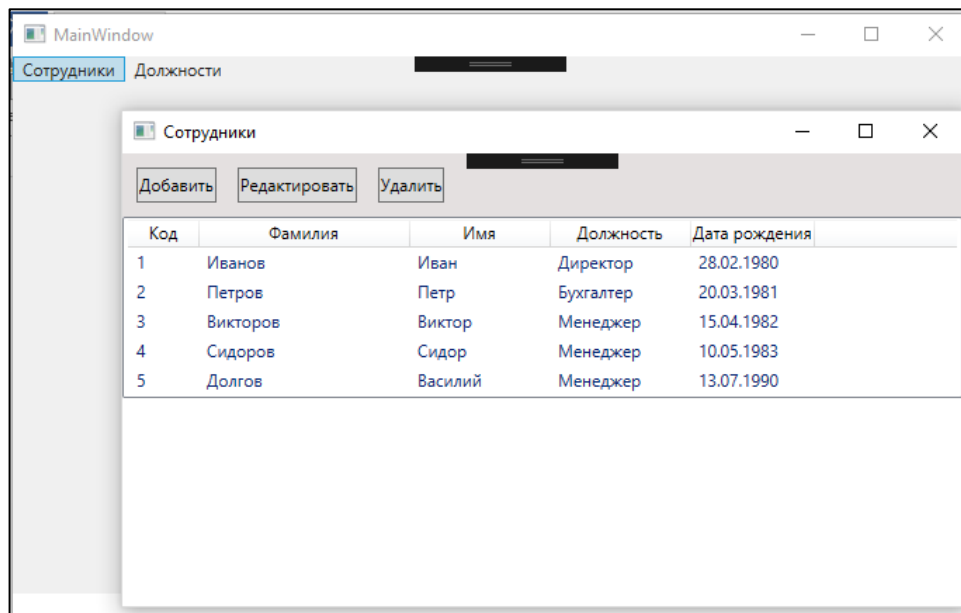


Рисунок 4.7 – Результат редактирования данных по сотруднику

Удаление сотрудника из списка сотрудников

При удалении сотрудника создадим для кнопки **Удалить** окна *WindowsEmployee* код обработчика *btnEdit_Click*.

```
private void btnDelete_Click(object sender, RoutedEventArgs e)
{
    PersonDPO person = (PersonDPO)lvEmployee.SelectedItem;
    if (person != null)
    {
        MessageBoxResult result = MessageBox.Show("Удалить данные по
сотруднику: \n" + person.LastName + " " + person.FirstName,
            "Предупреждение", MessageBoxButton.OKCancel, MessageBoxI-
mage.Warning);
        if (result == MessageBoxResult.OK)
        {
            // удаление данных в списке отображения данных
            personsDPO.Remove(person);
            // удаление данных в списке классов ListPerson<Person>
            Person per = new Person();
            per = per.CopyFromPersonDPO(person);
            vmPerson.ListPerson.Remove(per);
        }
    }
    else
    {
        MessageBox.Show("Необходимо выбрать данные по сотруднику для
удаления",
            "Предупреждение", MessageBoxButton.OK, MessageBoxI-
mage.Warning);
    }
}
```

При удалении данных по сотруднику формируем экземпляр *role* класса *Role* из выделенного элемента списка должностей *lvRole*.

```
Role role = (Role)lvRole.SelectedItem;
```

Если выбор должности для удаления произведен, то выводится предупреждение (рисунок 4.8).

```
MessageBoxResult result = MessageBox.Show("Удалить данные по  
должности: " +  
    role.NameRole, "Предупреждение", MessageBoxButton.OKCancel,  
    MessageBoxImage.Warning);
```

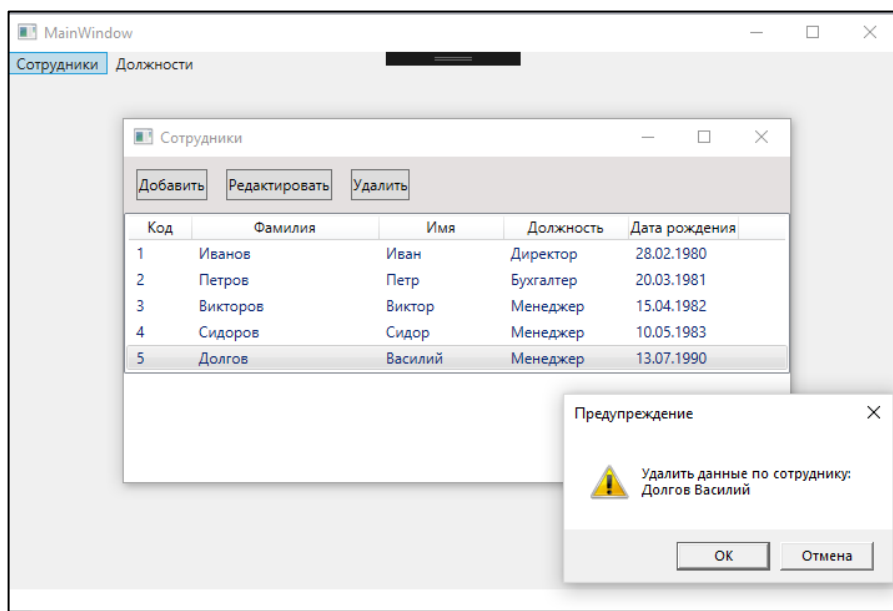


Рисунок 4.8 – Предупреждение об удалении данных по сотруднику

Если удаление подтверждается, то выбранный экземпляр сотрудника удаляется из списка сотрудников ListRole.

```
if (result == MessageBoxResult.OK)
{
    // удаление данных в списке отображения данных
    personsDPO.Remove(person);
    // удаление данных в списке классов ListPerson<Person>
    Person per = new Person();
    per = per.CopyFromPersonDPO(person);
    vmPerson.ListPerson.Remove(per);
}
```

Задание на лабораторную работу

1. Добавить в приложение функциональность для добавления, редактирования и удаление информации по классам модели данных в соответствии в варианте задания.
2. Протестировать добавленную функциональность.

ЛАБОРАТОРНАЯ РАБОТА 5. Разработка приложений по технологии MVVM

Цель работы: Получить навыки разработки приложений с использованием шаблона MVVM.

Общие сведения о шаблоне MVVM

При проектировании приложений WPF встают задачи обеспечения качественного дизайна программных компонентов, распределения функциональности между логическими слоями, удобства тестирования, сопровождения и модификации. Решению данных задач способствует шаблон проектирования «модель – представление – модель представления» – MVVP (model – view model – view), который обеспечивает создание архитектуры приложения, облегчающей чтение кода, удобство модификации и тестирования. Общий вид шаблона MVVP представлен на рисунке 5.1.

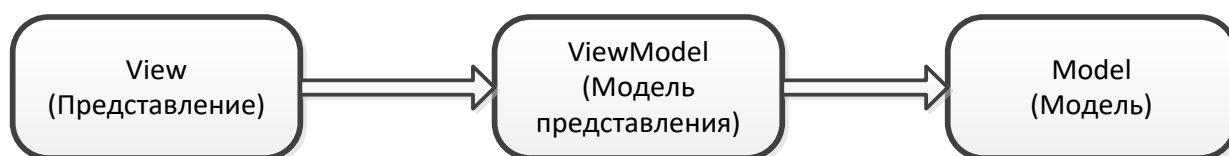


Рисунок 5.1 – Шаблон MVVP

Логический слой View представляет собой пользовательский интерфейс (UI – User Interface), декларативно заданный в виде XAML-файла. Данный архитектурный слой определяет представление приложения, которое обеспечивает взаимодействие пользователя с программной системой. Логический слой Model представляет собой модуль, реализующий бизнес-логику приложения (взаимодействие с источником данных, бизнес-правила). Модель представления ViewModel обеспечивает функциональность интерфейса, а также содержит логику управления состоянием UI и связь с моделью.

Шаблон MVVP использует:

- технологию связывания WPF (data bindings) для обеспечения гибкой связи между свойствами представления View и моделью представления ViewModel;
- модель команд (Commands) для обеспечения функциональности интерфейсных элементов контроля View в ViewModel;
- шаблоны данных, которые применяют представления к объектам модели представления;
- систему ресурсов для автоматического применения шаблонов данных во время выполнения приложения.

Такое архитектурное решение позволяет отделить процесс проектирования пользовательского интерфейса от бизнес-логики приложения. В идеальном случае представление View не зависит от реализации логики ViewModel и имеет только декларативное описание, то есть является XAML-документом. Модель представления ViewModel является абстракцией, которая представляет данные в объектах модели Model, нужных для отображения в представлении View. Для привязки объект модели представления устанавливается как контекст DataContext представления. Изменения свойств модели представления синхронизированы посредством привязки со свойствами представления. Когда пользователь инициирует определенное действие в пользовательском интерфейсе, то есть представлении (например, нажатие кнопки), для его выполнения активизируется команда в модели представления. Следует отметить, что все изменения данных модели Model всегда производит модель представления ViewModel, а не представление View. При этом классы представления не связаны непосредственно с классами модели, а модель представления и модель не зависят от конкретного представления. Модель на самом деле вообще не имеет представления о том, что существуют модель представления и представление.

Для изучения вопросов проектирования WPF-приложений на основе шаблона MVVM будем использовать приложение, разработанное в лабораторных работах 3 - 6. Сделаем копию проекта, в которую будем вносить изменения, которые соответствуют шаблону MVVM.

Модификация модели данных

Свойства, которые описывают данные, используемые в приложении, должны оповещать приложение о своем изменении. Для этого модели данных должны реализовывать интерфейс INotifyPropertyChanged, который включает описание события PropertyChanged и метод его генерации OnPropertyChanged().

Код класса Role.

```
using System.ComponentModel;
using System.Runtime.CompilerServices;
using WpfApplDemo2018.Annotations;
namespace WpfApplDemo2018.Model
{
    /// <summary>
    /// класс должность сотрудника
    /// </summary>
    public class Role: INotifyPropertyChanged
    {
        /// <summary>
        /// код должности
        /// </summary>
        public int Id { get; set; }
        /// <summary>
        /// наименование должности
        /// </summary>
```

```

private string nameRole;
/// <summary>
/// наименование должности
/// </summary>
public string NameRole
{
    get { return nameRole; }
    set
    {
        nameRole = value;
        OnPropertyChanged("NameRole");
    }
}
public Role() { }
public Role(int id, string nameRole)
{
    this.Id = id;
    this.NameRole = nameRole;
}
/// <summary>
/// Метод поверхностного копирования
/// </summary>
/// <returns></returns>
public Role ShallowCopy()
{
    return (Role)this.MemberwiseClone();
}
public event PropertyChangedEventHandler PropertyChanged;
[NotifyPropertyChangedInvoker]
protected virtual void OnPropertyChanged([CallerMemberName]
string propertyName = "")
{
    PropertyChanged?.Invoke(this, new PropertyChangedEven-
tArgs(propertyName));
}
}

```

В определении класса Role добавим наследование его от интерфейса **INotifyPropertyChanged**.

```
public class Role: INotifyPropertyChanged
```

Реализация интерфейса включает событие и метод его генерации.

```

public event PropertyChangedEventHandler PropertyChanged;
[NotifyPropertyChangedInvoker]
protected virtual void OnPropertyChanged([CallerMemberName]
string propertyName = "")
{
    PropertyChanged?.Invoke(this, new PropertyChangedEven-
tArgs(propertyName));
}

```

Свойства, которые должны информировать приложение при своем изменении, должны генерировать событие **PropertyChanged**.

```

public string NameRole
{
    get { return nameRole; }
    set
    {
        nameRole = value;
        OnPropertyChanged("NameRole");
    }
}

```

Аналогичные изменения введем в класс PersonDpo.

Код класса PersonDpo.

```

using System;
using System.ComponentModel;
using System.Runtime.CompilerServices;
using WpfApplDemo2018.Annotations;
using WpfApplDemo2018.ViewModel;
namespace WpfApplDemo2018.Model
{
    /// <summary>
    /// класс отображения данных по сотруднику
    /// </summary>
    public class PersonDpo: INotifyPropertyChanged
    {
        /// <summary>
        /// код сотрудника
        /// </summary>
        public int Id { get; set; }
        /// <summary>
        /// должность сотрудника
        /// </summary>
        private string _roleName;
        /// <summary>
        /// должность сотрудника
        /// </summary>
        public string RoleName
        {
            get { return _roleName; }
            set
            {
                _roleName = value;
                OnPropertyChanged("RoleName");
            }
        }
        /// <summary>
        /// имя сотрудника
        /// </summary>
        private string firstName;
        /// <summary>
        /// имя сотрудника
        /// </summary>
        public string FirstName
        {

```



```

    get { return firstName; }
    set
    {
        firstName = value;
        OnPropertyChanged("FirstName");
    }
}
/// <summary>
/// фамилия сотрудника
/// </summary>
private string lastName;
/// <summary>
/// фамилия сотрудника
/// </summary>
public string LastName
{
    get { return lastName; }
    set
    {
        lastName = value;
        OnPropertyChanged("LastName");
    }
}
/// <summary>
/// дата рождения сотрудника
/// </summary>
private DateTime birthday;
/// <summary>
/// дата рождения сотрудника
/// </summary>
public DateTime Birthday
{
    get { return birthday; }
    set
    {
        birthday = value;
        OnPropertyChanged("Birthday");
    }
}
public PersonDpo() { }
public PersonDpo(int id, string roleName, string firstName,
string lastName, DateTime birthday)
{
    this.Id = id;
    this.RoleName = roleName;
    this.FirstName = firstName;
    this.LastName = lastName;
    this.Birthday = birthday;
}
/// <summary>
/// Метод поверхностного копирования
/// </summary>

```

```

    /// <returns></returns>
    public PersonDpo ShallowCopy()
    {
        return (PersonDpo)this.MemberwiseClone();
    }
    /// <summary>
    /// копирование данных из класса Person
    /// </summary>
    /// <param name="person"></param>
    /// <returns></returns>
    public PersonDpo CopyFromPerson(Person person)
    {
        PersonDpo perDpo = new PersonDpo();
        RoleViewModel vmRole = new RoleViewModel();
        string role = string.Empty;
        foreach (var r in vmRole.ListRole)
        {
            if (r.Id == person.RoleId)
            {
                role = r.NameRole;
                break;
            }
        }
        if (role != string.Empty)
        {
            perDpo.Id = person.Id;
            perDpo.RoleName = role;
            perDpo.FirstName = person.FirstName;
            perDpo.LastName = person.LastName;
            perDpo.Birthday = person.Birthday;
        }
        return perDpo;
    }
    public event PropertyChangedEventHandler PropertyChanged;
    [NotifyPropertyChangedInvocator]
    protected virtual void OnPropertyChanged([CallerMemberName]
string propertyName = "")
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEven-
tArgs(propertyName));
    }
}

```

В класс RoleViewModel введем следующие изменения:

1. Данный класс должен наследовать интерфейс INotifyPropertyChanged

```
public class RoleViewModel: INotifyPropertyChanged
```

2. В классе реализуем члены интерфейса INotifyPropertyChanged

```

public event PropertyChangedEventHandler PropertyChanged;
protected virtual void OnPropertyChanged([CallerMemberName]
string propertyName = "")

```

```

    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(
            propertyName));
    }
}

```

3. Добавим свойство `SelectedRole`, которому присваивается значение должности, выбранной в списке должностей.

```

public Role SelectedRole
{
    get
    {
        return selectedRole;
    }
    set
    {
        selectedRole = value;
        OnPropertyChanged("SelectedRole");
        EditRole.CanExecute(true);
    }
}

```

Код класса `RoleViewModel`

```

using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Windows;
using WpfApplDemo2018.Helper;
using WpfApplDemo2018.Model;
using WpfApplDemo2018.View;
namespace WpfApplDemo2018.ViewModel
{
    public class RoleViewModel: INotifyPropertyChanged
    {
        /// <summary>
        /// выбранная в списке должность
        /// </summary>
        private Role selectedRole;
        /// <summary>
        /// выбранная в списке должность
        /// </summary>
        public Role SelectedRole
        {
            get
            {
                return selectedRole;
            }
            set
            {
                selectedRole = value;
                OnPropertyChanged("SelectedRole");
                EditRole.CanExecute(true);
            }
        }
    }
}

```

```

    }
}
/// <summary>
/// коллекция должностей сотрудников
/// </summary>
public ObservableCollection<Role> ListRole { get; set; } = new
ObservableCollection<Role>();
public RoleViewModel()
{
    this.ListRole.Add(new Role
    {
        Id = 1,
        NameRole = "Директор"
    });
    this.ListRole.Add(new Role
    {
        Id = 2,
        NameRole = "Бухгалтер"
    });
    this.ListRole.Add(new Role
    {
        Id = 3,
        NameRole = "Менеджер"
    });
}
/// <summary>
/// Нахождение максимального Id в коллекции
/// </summary>
/// <returns></returns>
public int MaxId()
{
    int max = 0;
    foreach (var r in this.ListRole)
    {
        if (max < r.Id)
        {
            max = r.Id;
        }
    }
    return max;
}
public event PropertyChangedEventHandler PropertyChanged;
protected virtual void OnPropertyChanged([CallerMemberName]
string propertyName = "")
{
    PropertyChanged?.Invoke(this, new PropertyChangedEven-
tArgs(propertyName));
}
}
}

```

В конструкторе класса WindowRole зададим свойству DataContext значение созданного экземпляра класса RoleViewModel.

```
public WindowRole()
{
    InitializeComponent();
    DataContext = new RoleViewModel();
}
```

Задание контексту данных окна WindowRole значения экземпляра класса RoleViewModel позволит ссылаться в окне WindowRole на свойства класса RoleViewModel. На данном этапе для отображения данных в окне WindowRole для приложения необходимо иметь доступ к свойству ListRole, которое описывает коллекцию данных по должностям сотрудников. Для отображения данных по должностям сотрудников необходимо в списке ListView связать источник данных ItemsSource с свойством ListRole.

```
<ListView ItemsSource="{Binding ListRole}"
    Background="#FFD3CDCD">
```

Для тестирования проведенных в приложении изменений проведем сборку приложения и запустим его на выполнение. При выборе пункта меню Должности, окно отображения должностей будет иметь вид приведенный на рисунке 5.2.

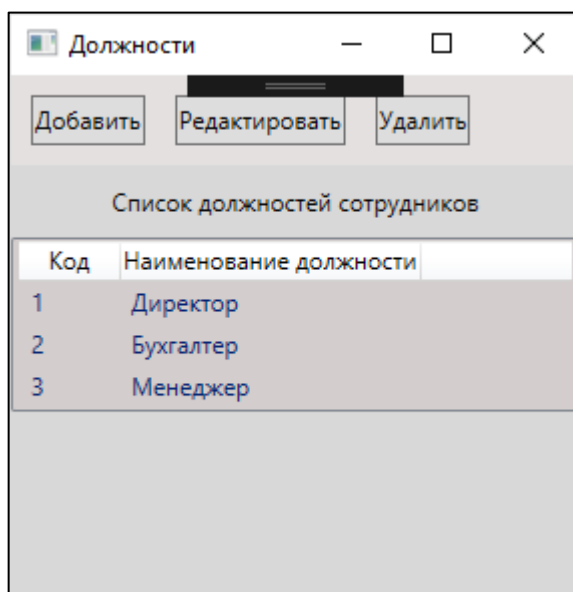


Рисунок 5.2 – Окно вывода данных по должностям

Создание модели команд

Для взаимодействия пользователя и приложения в MVVM используются команды. Это не значит, что вовсе не стоит использовать события и событийную модель, однако везде, где возможно, вместо событий следует использовать команды.

Для использования команд добавим в папку Helper новый класс RelayCommand, который поддерживает интерфейс ICommand.

```
using System;
using System.Windows.Input;
namespace WpfApplDemo2018.Helper
{
    public class RelayCommand: ICommand
```

```

{
    private Action<object> execute;
    private Func<object, bool> canExecute;
    public event EventHandler CanExecuteChanged
    {
        add { CommandManager.RequerySuggested += value; }
        remove { CommandManager.RequerySuggested -= value; }
    }
    public RelayCommand(Action<object> execute, Func<object, bool>
canExecute = null)
    {
        this.execute = execute;
        this.canExecute = canExecute;
    }
    public bool CanExecute(object parameter)
    {
        return this.canExecute == null || this.canExecute(parameter);
    }
    public void Execute(object parameter)
    {
        this.execute(parameter);
    }
}
}

```

Класс RelayCommand реализует два метода и событие интерфейса ICommand:

- CanExecute: определяет, может ли команда выполняться;
- Execute: собственно выполняет логику команды;
- CanExecuteChanged: событие, которое вызывается при изменении условий, указывающий, может ли команда выполняться. Для этого используется событие CommandManager.RequerySuggested.

Ключевым является метод Execute. Для его выполнения в конструкторе команды передается делегат типа Action<object> и при необходимости делегат типа Func<object>. При этом класс команды не знает какое именно действие будет выполняться.

Команда добавления новой должности AddRole

При добавлении новой должности необходимо добавить в список должностей новый экземпляр класса Role. Создадим в классе RoleViewModel свойство AddRole типа RelayCommand. При создании команды ей в качестве параметра передается делегат типа Action<object>, который реализует функциональность добавления новой должности в список должностей. Код добавления новой должности аналогичен коду, разработанному в лабораторной работе 5.

```

/// команда добавления новой должности
private RelayCommand addRole;
public RelayCommand AddRole
{
    get
    {
        return addRole ??
            (addRole = new RelayCommand(obj =>
            {
                WindowNewRole wnRole = new WindowNewRole
                {
                    Title = "Новая должность",
                };
                // формирование кода новой должности
                int maxIdRole = MaxId() + 1;
                Role role = new Role{Id = maxIdRole};
                wnRole.DataContext = role;
                if (wnRole.ShowDialog() == true)
                {
                    ListRole.Add(role);
                }
                SelectedRole = role;
            }));
    }
}

```

Для вызова команды AddRole необходимо изменить XAML-код для кнопки Добавить, связав свойство Command со свойством AddRole класса RoleViewModel.

```

<Button Content="Добавить" Height="25" Margin="10,10,5,10"
        Command="{Binding AddRole}"/>

```

Для проверки функциональности приложения в части добавления новой должности в список должностей сотрудников запустим на выполнение приложение и в окне Должности нажмем кнопку Добавить. В окне Новая должность заполним поле Должность и нажмем кнопку Сохранить (рисунок 5.3).

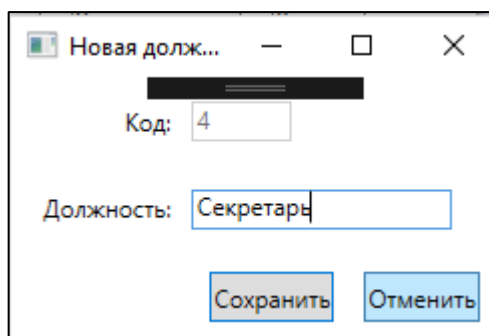


Рисунок 5.3 – Добавление новой должности

Результатом добавления новой должности будет изменение в списке должностей (рисунок 5.4).

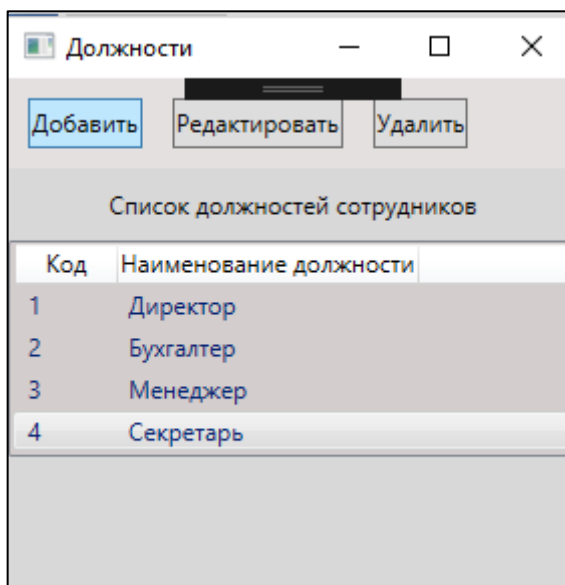


Рисунок 5.4 – Добавление новой должности в список должностей

Команда редактирования должности EditRole

При редактировании данных по существующей должности необходимо выделить в списке должностей строку с наименованием должности для редактирования. Создадим в классе RoleViewModel свойство EditRole типа RelayCommand. При создании команды ей в качестве параметра передается делегат типа Action<object>, который реализует функциональность добавления новой должности в список должностей, и делегат типа Func<object>. Код редактирования должности аналогичен коду, разработанному в лабораторной работе 5.

```
private RelayCommand editRole;
public RelayCommand EditRole
{
    get
    {
        return editRole ??
        (editRole = new RelayCommand(obj =>
        {
            WindowNewRole wnRole = new WindowNewRole
            { Title = "Редактирование должности", };
            Role role = SelectedRole;
            Role tempRole = new Role();
            tempRole = role.ShallowCopy();
            wnRole.DataContext = tempRole;
            if (wnRole.ShowDialog() == true)
            {
                // сохранение данных в оперативной памяти
                role.NameRole = tempRole.NameRole;
            }
        }, (obj) => SelectedRole != null && ListRole.Count > 0));
    }
}
```

При создании экземпляра команды EditRole конструктору класса RelayCommand в качестве второго параметра передается делегат Func<object> в

виде лямбда-выражения (obj) => SelectedRole != null && ListRole.Count > 0, которое определяет доступность команды. Команда редактирования доступна, когда в списке должностей присутствует выделенный элемент (SelectedRole != null) и список должностей не пустой (ListRole.Count > 0).

Для вызова команды EditRole необходимо изменить XAML-код для кнопки Редактировать, связав свойство Command со свойством EditRole класса RoleViewModel.

```
<Button Content="Редактировать" Height="25" Margin="10,10,5,10"
        Command="{Binding EditRole}"/>
```

Для проверки функциональности приложения в части редактирования существующие должности выделим в списке должностей строку с должностью Секретарь. В окне Редактирование должности исправим наименование должности на Секретарь директора (рисунок 5.5).

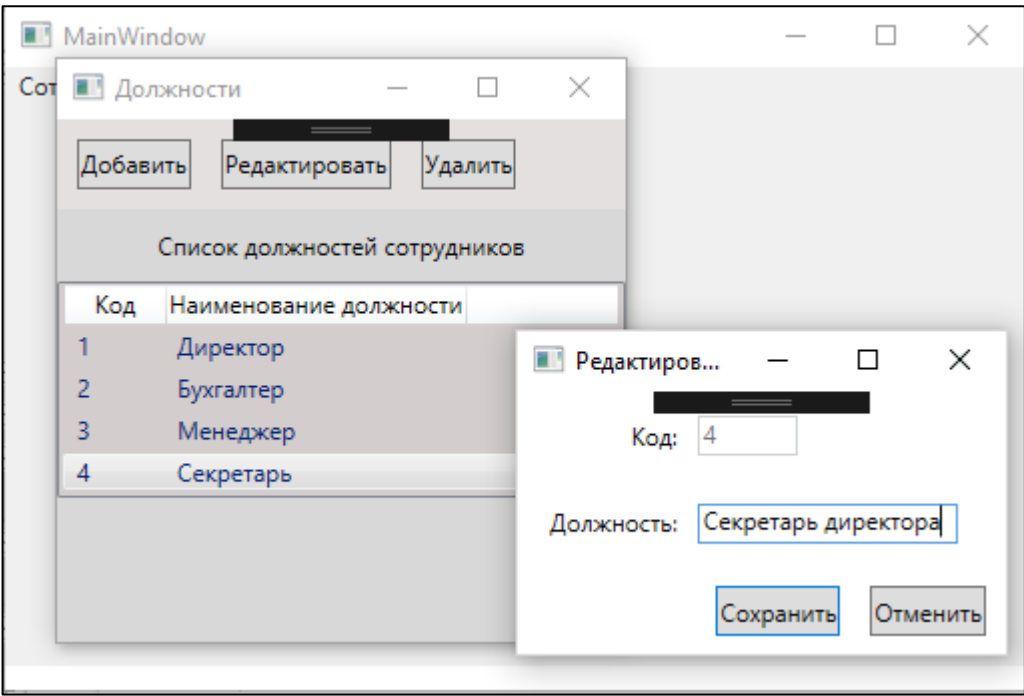


Рисунок 5.5 – Редактирование наименования должности

После нажатия кнопки Сохранить будет выведен список должностей с отредактированным наименованием должности Секретарь (рисунок 5.6).

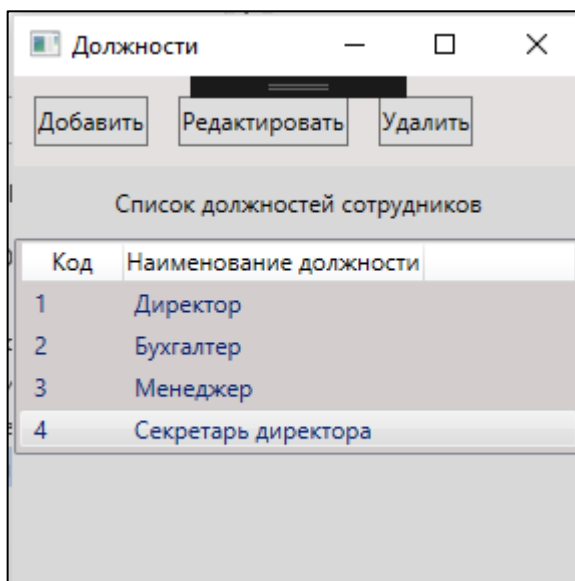


Рисунок 5.6 – Результат редактирования наименования должности

Команда удаления должности DeleteRole

При удалении данных по существующей должности необходимо выделить в списке должностей строку с наименованием должности для удаления. Создадим в классе RoleViewModel свойство DeleteRole типа RelayCommand. При создании команды ей в качестве параметра передается делегат типа Action<object>, который реализует функциональность удаления должности в списке должностей, и делегат типа Func<object>, который определяет доступность команды. Код удаления должности аналогичен коду, разработанному в лабораторной работе 5.

```
private RelayCommand deleteRole;
public RelayCommand DeleteRole
{
    get
    {
        return deleteRole ??
            (deleteRole = new RelayCommand(obj =>
            {
                Role role = SelectedRole;
                MessageBoxResult result = MessageBox.Show("Удалить данные по
должности: " +
                role.NameRole, "Предупреждение", MessageBoxButton.OKCancel,
                MessageBoxImage.Warning);
                if (result == MessageBoxResult.OK)
                {
                    ListRole.Remove(role);
                }
            }, (obj) => SelectedRole != null && ListRole.Count > 0));
    }
}
```

Для вызова команды DeleteRole необходимо изменить XAML-код для кнопки Удалить, связав свойство Command со свойством DeleteRole класса RoleViewModel.

```
<Button Content="Удалить" Height="25" Margin="10,10,5,10"
        Command="{Binding DeleteRole}"/>
```

Для проверки функциональности приложения в части удаления существующей должности выделим в списке должностей строку с должностью Секретарь директора. После нажатия кнопки Удалить будет выведено предупреждение (рисунок 5.7).

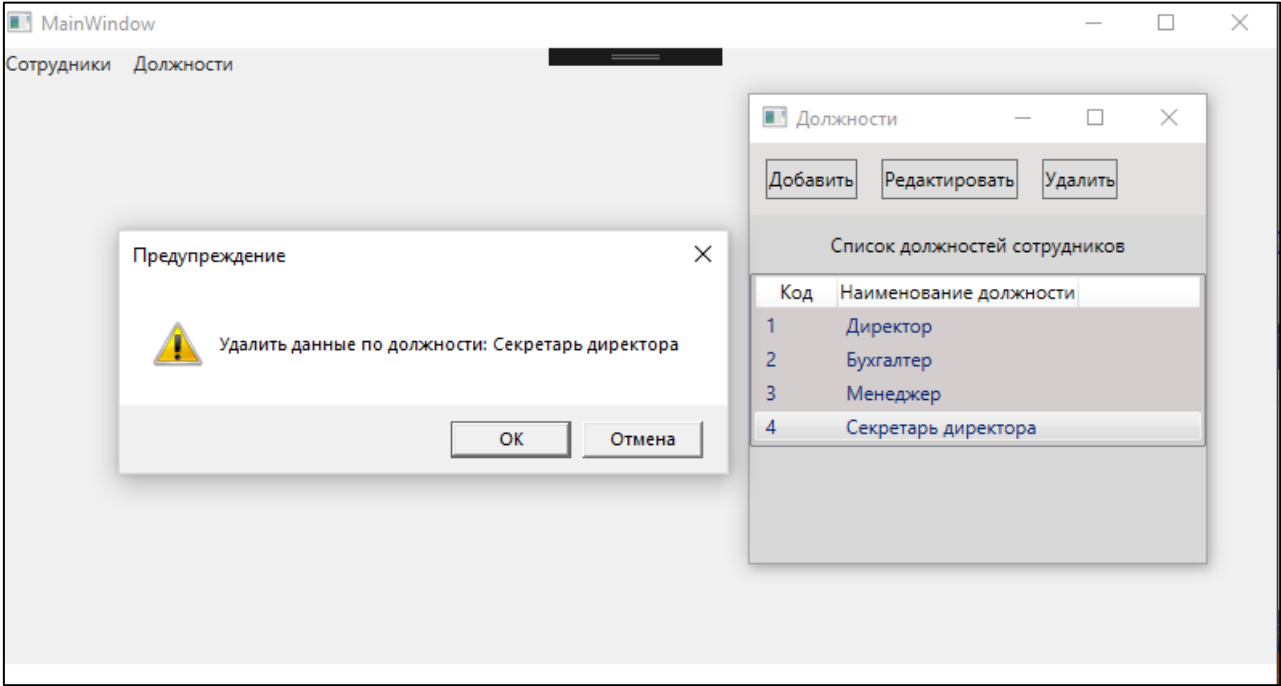


Рисунок 5.7 – Удаление должности

Если в диалоговом окне будет подтверждено удаление (нажатие кнопки ОК), то должность будет удалена из списка должностей (рисунок 5.8).

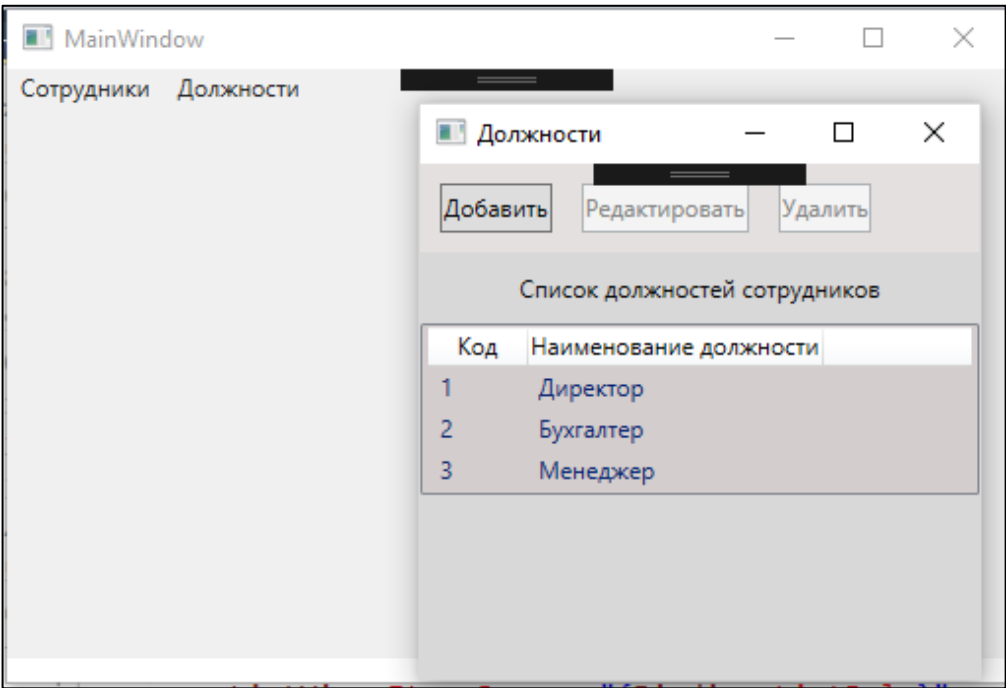


Рисунок 5.8 – Результат удаления должности

Обратите внимание на то, что кнопки Редактировать и Удалить неактивны, так как в списке не выделена ни одна строка с должностью.

После создания команд и свойств в классе `RoleViewModel` в коде класса `WindowRole` в конструкторе осуществляется только инициализация компонентов и задание контекста данных.

```
public partial class WindowRole : Window
{
    public WindowRole()
    {
        InitializeComponent();
        DataContext = new RoleViewModel();
    }
}
```

Бизнес-логика приложения практически полностью перенесена в класс `RoleViewModel`, что способствует упрощению тестирования приложения и отделению презентационного уровня (XAML-описание) от бизнес-логики.

Модификация кода приложения для работы с данными по сотрудникам

Внесем изменения в модель данных по сотруднику, которые отображаются в приложении.

```
public class PersonDpo: INotifyPropertyChanged
{
    /// <summary>
    /// код сотрудника
    /// </summary>
    public int Id { get; set; }
    /// <summary>
    /// должность сотрудника
    /// </summary>
    private string _roleName;
    /// <summary>
    /// должность сотрудника
    /// </summary>
    public string RoleName
    {
        get { return _roleName; }
        set
        {
            _roleName = value;
            OnPropertyChanged("RoleName");
        }
    }
    /// <summary>
    /// имя сотрудника
    /// </summary>
    private string firstName;
    /// <summary>
    /// имя сотрудника
    /// </summary>
```

```

public string FirstName
{
    get { return firstName; }
    set
    {
        firstName = value;
        OnPropertyChanged("FirstName");
    }
}
/// <summary>
/// фамилия сотрудника
/// </summary>
private string lastName;
/// <summary>
/// фамилия сотрудника
/// </summary>
public string LastName
{
    get { return lastName; }
    set
    {
        lastName = value;
        OnPropertyChanged("LastName");
    }
}
/// <summary>
/// дата рождения сотрудника
/// </summary>
private DateTime birthday;
/// <summary>
/// дата рождения сотрудника
/// </summary>
public DateTime Birthday
{
    get { return birthday; }
    set
    {
        birthday = value;
        OnPropertyChanged("Birthday");
    }
}
public PersonDpo() { }
public PersonDpo(int id, string roleName, string firstName,
string lastName, DateTime birthday)
{
    this.Id = id;
    this.RoleName = roleName;
    this.FirstName = firstName;
    this.LastName = lastName;
    this.Birthday = birthday;
}
/// <summary>

```

```

    /// Метод поверхностного копирования
    /// </summary>
    /// <returns></returns>
    public PersonDpo ShallowCopy()
    {
        return (PersonDpo)this.MemberwiseClone();
    }
    /// <summary>
    /// копирование данных из класса Person
    /// </summary>
    /// <param name="person"></param>
    /// <returns></returns>
    public PersonDpo CopyFromPerson(Person person)
    {
        PersonDpo perDpo = new PersonDpo();
        RoleViewModel vmRole = new RoleViewModel();
        string role = string.Empty;
        foreach (var r in vmRole.ListRole)
        {
            if (r.Id == person.RoleId)
            {
                role = r.NameRole;
                break;
            }
        }
        if (role != string.Empty)
        {
            perDpo.Id = person.Id;
            perDpo.RoleName = role;
            perDpo.FirstName = person.FirstName;
            perDpo.LastName = person.LastName;
            perDpo.Birthday = person.Birthday;
        }
        return perDpo;
    }
    public event PropertyChangedEventHandler PropertyChanged;
    [NotifyPropertyChangedInvocator]
    protected virtual void OnPropertyChanged([CallerMemberName]
string propertyName = "")
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEven-
tArgs(propertyName));
    }
}

```

Внесем изменения в класс PersonViewModel/

```

public class PersonViewModel: INotifyPropertyChanged
{
    private PersonDpo selectedPersonDpo;
    /// <summary>
    /// выделенные в списке данные по сотруднику
    /// </summary>
    public PersonDpo SelectedPersonDpo

```

```

{
    get { return selectedPersonDpo; }
    set
    {
        selectedPersonDpo = value;
        OnPropertyChanged("SelectedPersonDpo");
    }
}
/// <summary>
/// коллекция данных по сотрудникам
/// </summary>
public ObservableCollection<Person> ListPerson { get; set; } =
new ObservableCollection<Person>();
public ObservableCollection<PersonDpo> ListPersonDpo { get;
set; } = new ObservableCollection<PersonDpo>();
public PersonViewModel()
{
    this.ListPerson.Add(
        new Person
        {
            Id = 1,
            RoleId = 1,
            FirstName = "Иван",
            LastName = "Иванов",
            Birthday = new DateTime(1980,02,28)
        });
    this.ListPerson.Add(
        new Person
        {
            Id = 2,
            RoleId = 2,
            FirstName = "Петр",
            LastName = "Петров",
            Birthday = new DateTime(1981,03,20)
        });
    this.ListPerson.Add(
        new Person
        {
            Id = 3,
            RoleId = 3,
            FirstName = "Виктор",
            LastName = "Викторов",
            Birthday = new DateTime(1982,04,15)
        });
    this.ListPerson.Add(
        new Person
        {
            Id = 4,
            RoleId = 3,
            FirstName = "Сидор",
            LastName = "Сидоров",
            Birthday = new DateTime(1983,05,10)
        });
}

```

```

    });
    ListPersonDpo = GetListPersonDpo();
}
public ObservableCollection<PersonDpo> GetListPersonDpo()
{
    foreach (var person in ListPerson)
    {
        PersonDpo p = new PersonDpo();
        p = p.CopyFromPerson(person);
        ListPersonDpo.Add(p);
    }
    return ListPersonDpo;
}
/// <summary>
/// Нахождение максимального Id в коллекции данных
/// </summary>
/// <returns></returns>
public int MaxId()
{
    int max = 0;
    foreach (var r in this.ListPerson)
    {
        if (max < r.Id)
        {
            max = r.Id;
        }
    }
    return max;
}
#region AddPerson
/// <summary>
/// добавление сотрудника
/// </summary>
private RelayCommand addPerson;
/// <summary>
/// добавление сотрудника
/// </summary>
public RelayCommand AddPerson
{
    get
    {
        return addPerson ??
            (addPerson = new RelayCommand(obj =>
            {
                WindowNewEmployee wnPerson = new WindowNewEmployee
                {
                    Title = "Новый сотрудник"
                };
                // формирование кода нового сотрудника
                int maxIdPerson = MaxId() + 1;
                PersonDpo per = new PersonDpo
                {

```



```

        Id = maxIdPerson,
        Birthday = DateTime.Now
    };
    wnPerson.DataContext = per;
    if (wnPerson.ShowDialog() == true)
    {
        Role r = (Role)wnPerson.CbRole.SelectedValue;
        per.RoleName = r.NameRole;
        ListPersonDpo.Add(per);
        // добавление нового сотрудника в коллекцию ListPer-
son<Person>
        Person p = new Person();
        p = p.CopyFromPersonDPO(per);
        ListPerson.Add(p);
    }
},
(obj) => true));
}
}
#endregion
#region EditPerson
/// команда редактирования данных по сотруднику
private RelayCommand editPerson;
public RelayCommand EditPerson
{
    get
    {
        return editPerson ??
            (editPerson = new RelayCommand(obj =>
            {
                WindowNewEmployee wnPerson = new WindowNewEmployee()
                {
                    Title = "Редактирование данных сотрудника",
                };
                PersonDpo personDpo = SelectedPersonDpo;
                PersonDpo tempPerson = new PersonDpo();
                tempPerson = personDpo.ShallowCopy();
                wnPerson.DataContext = tempPerson;

                //wnPerson.CbRole.ItemsSource = new ListRole();
                if (wnPerson.ShowDialog() == true)
                {
                    // сохранение данных в оперативной памяти
                    // перенос данных из временного класса в класс отображения
                    // данных
                    Role r = (Role)wnPerson.CbRole.SelectedValue;
                    personDpo.RoleName = r.NameRole;
                    personDpo.FirstName = tempPerson.FirstName;
                    personDpo.LastName = tempPerson.LastName;
                    personDpo.Birthday = tempPerson.Birthday;
                    // перенос данных из класса отображения данных в класс Person
                    FindPerson finder = new FindPerson(personDpo.Id);

```

```

        List<Person> listPerson = ListPerson.ToList();
        Person p = listPerson.Find(new Predicate<Person>(finder.PersonPredicate));
        p = p.CopyFromPersonDPO(personDpo);
    }
    }, (obj) => SelectedPersonDpo != null && ListPersonDpo.Count > 0));
    }
}
#endregion
#region DeletePerson
/// команда удаления данных по сотруднику
private RelayCommand deletePerson;
public RelayCommand DeletePerson
{
    get
    {
        return deletePerson ??
            (deletePerson = new RelayCommand(obj =>
            {
                PersonDpo person = SelectedPersonDpo;
                MessageBoxResult result = MessageBox.Show("Удалить
данные по сотруднику: \n" + person.LastName + " " + person.FirstName,
                "Предупреждение", MessageBoxButton.OKCancel, MessageBoxImage.Warning);
                if (result == MessageBoxResult.OK)
                {
                    // удаление данных в списке отображения данных
                    ListPersonDpo.Remove(person);
                    // удаление данных в списке классов ListPerson<Person>
                    Person per = new Person();
                    per = per.CopyFromPersonDPO(person);
                    ListPerson.Remove(per);
                }
            }, (obj) => SelectedPersonDpo != null && ListPersonDpo.Count > 0));
            }
    }
}
#endregion
public event PropertyChangedEventHandler PropertyChanged;
[NotifyPropertyChangedInvocator]
protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = "")
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
}
}

```

Внесем изменения в XAML-описание кнопок окна WindowsEmployee.

```

<StackPanel Orientation="Horizontal" Background="#FFE4E0E0">
    <Button Content="Добавить" Height="25" Margin="10,10,5,10"

```

```

        Command="{Binding AddPerson}"/>
<Button Margin="10,10,5,10" Content="Редактировать" Height="25"
        Command="{Binding EditPerson}"/>
<Button Content="Удалить" Height="25" Margin="10,10,5,10"
        Command="{Binding DeletePerson}"/>
</StackPanel>

```

Отредактируем код класса WindowEmployee.

```

public partial class WindowEmployee : Window
{
    public WindowEmployee()
    {
        InitializeComponent();
        DataContext = new PersonViewModel();
    }
}

```

При запуске приложения и выборе пункта меню Сотрудники будет выведен список сотрудников в окне Сотрудники (рисунок 5.9).

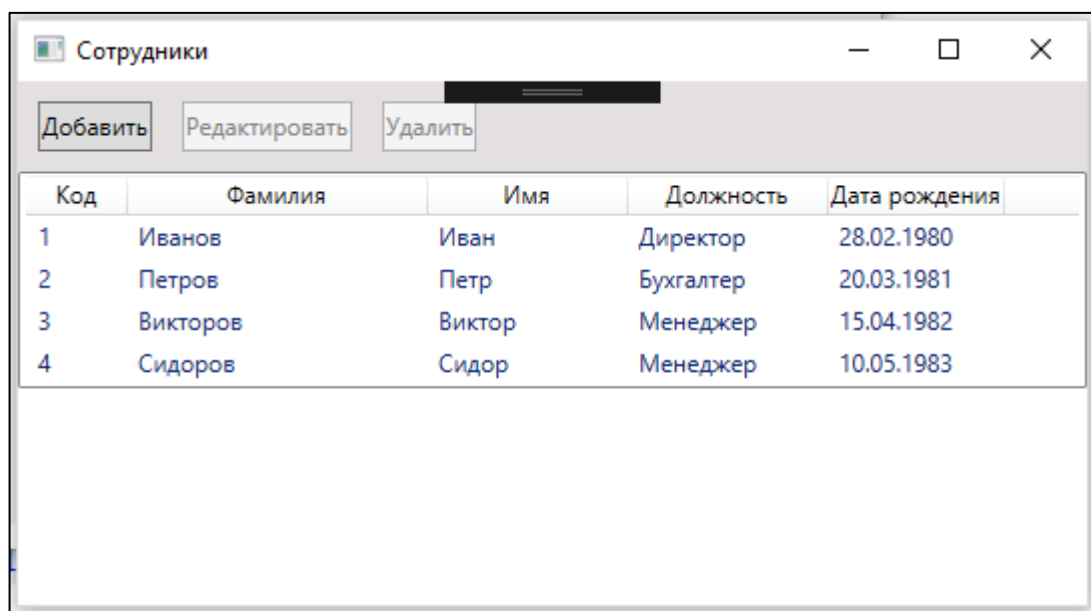


Рисунок 5.9 – Список сотрудников

В окне Сотрудники активной является кнопка Добавить, а кнопки Редактировать и Удалить неактивны, так как в списке не выделена ни одна строка.

При добавлении данных по сотруднику выводится окно Новый сотрудник, в котором необходимо ввести данные о сотруднике (рисунок 5.10).

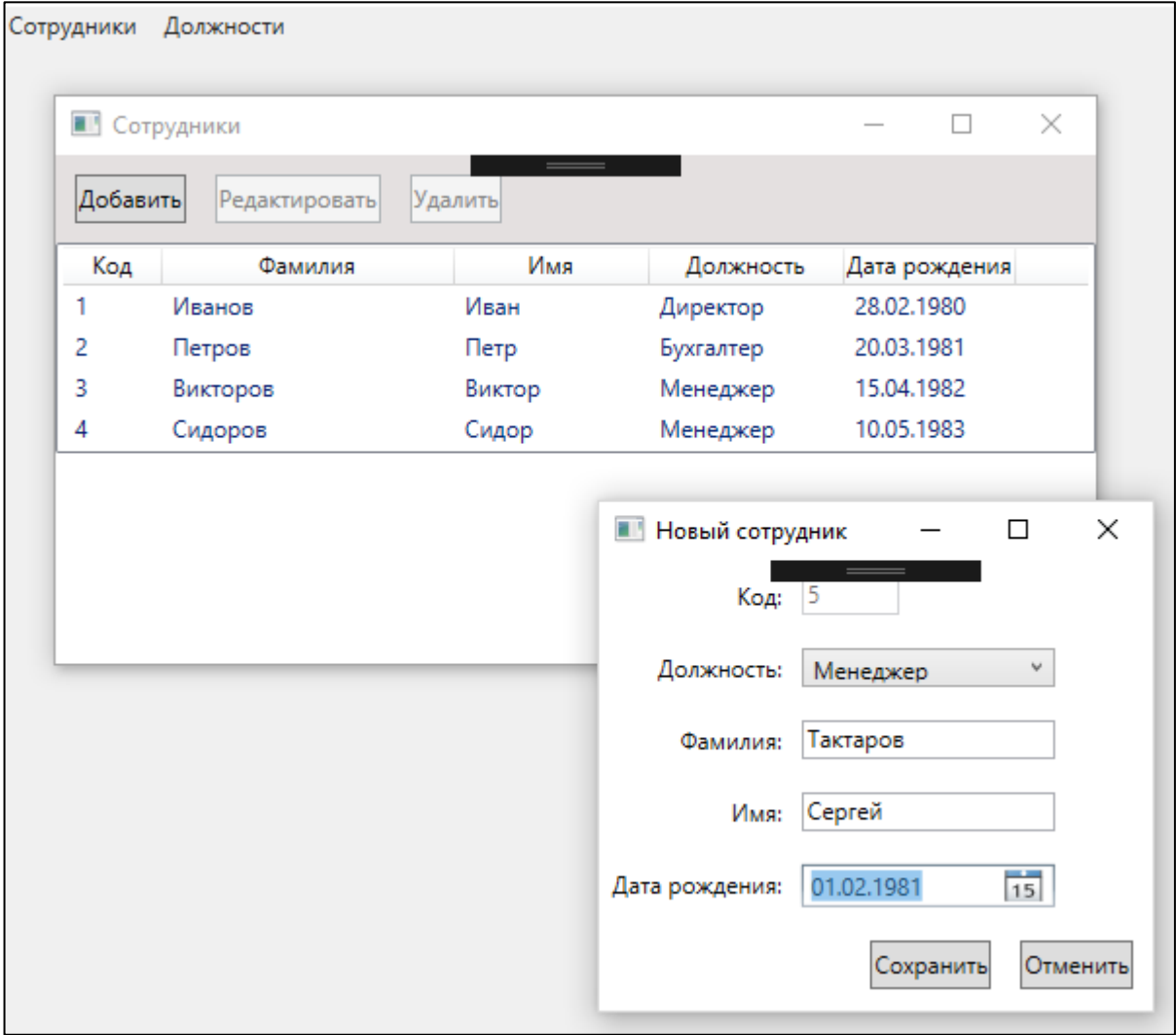


Рисунок 5.10 – Добавление нового сотрудника

При выборе для редактирования строки с данными по сотруднику Викторов кнопки Редактировать и Удалить становятся активными. После нажатия кнопки Редактировать открывается окно редактирования (рисунок 5.11), в котором можно изменить данные по сотруднику, кроме кода.

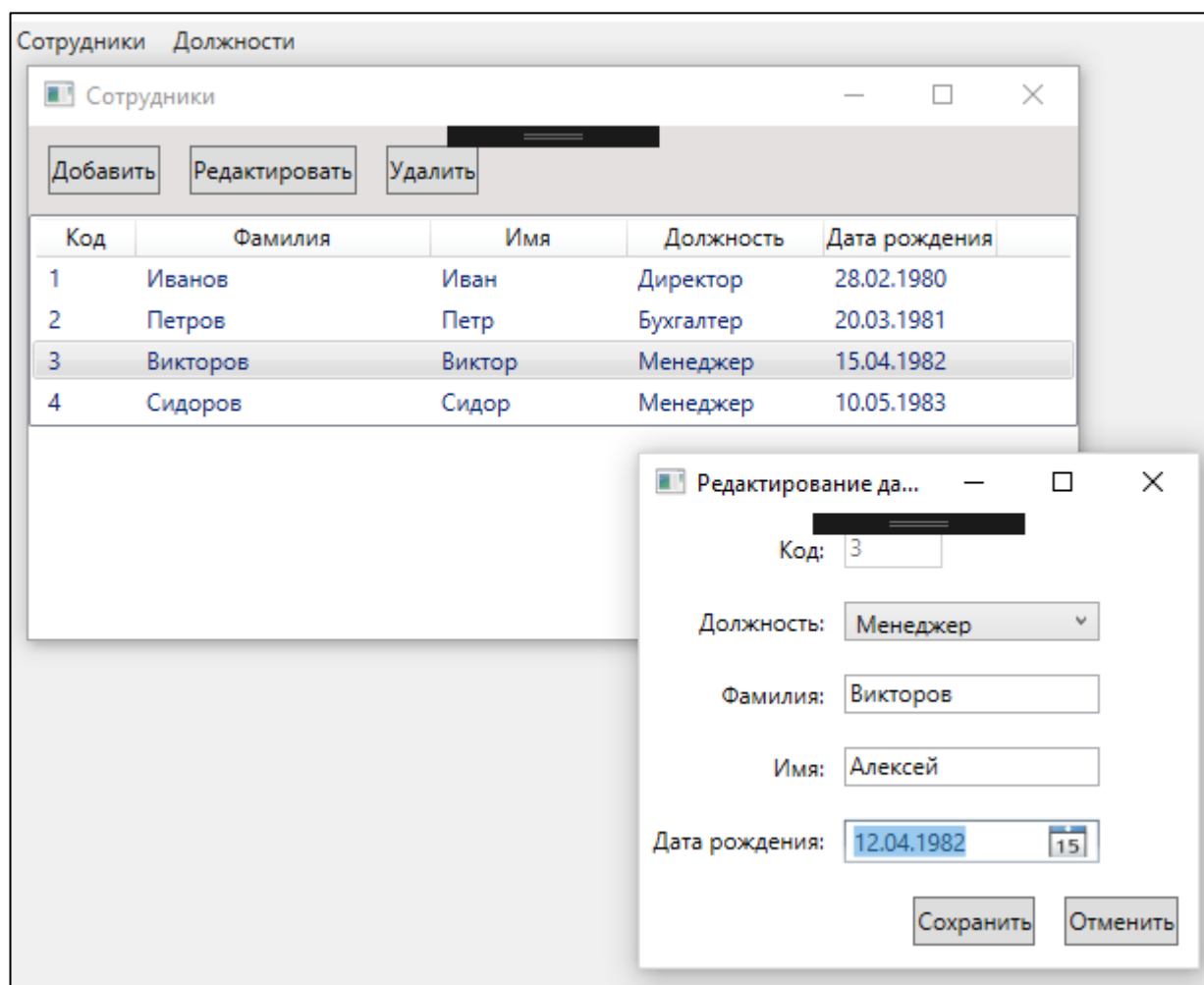


Рисунок 5.11 – Редактирование данных по сотруднику

При удалении данных по сотруднику выводится предупреждение (рисунок 5.12).

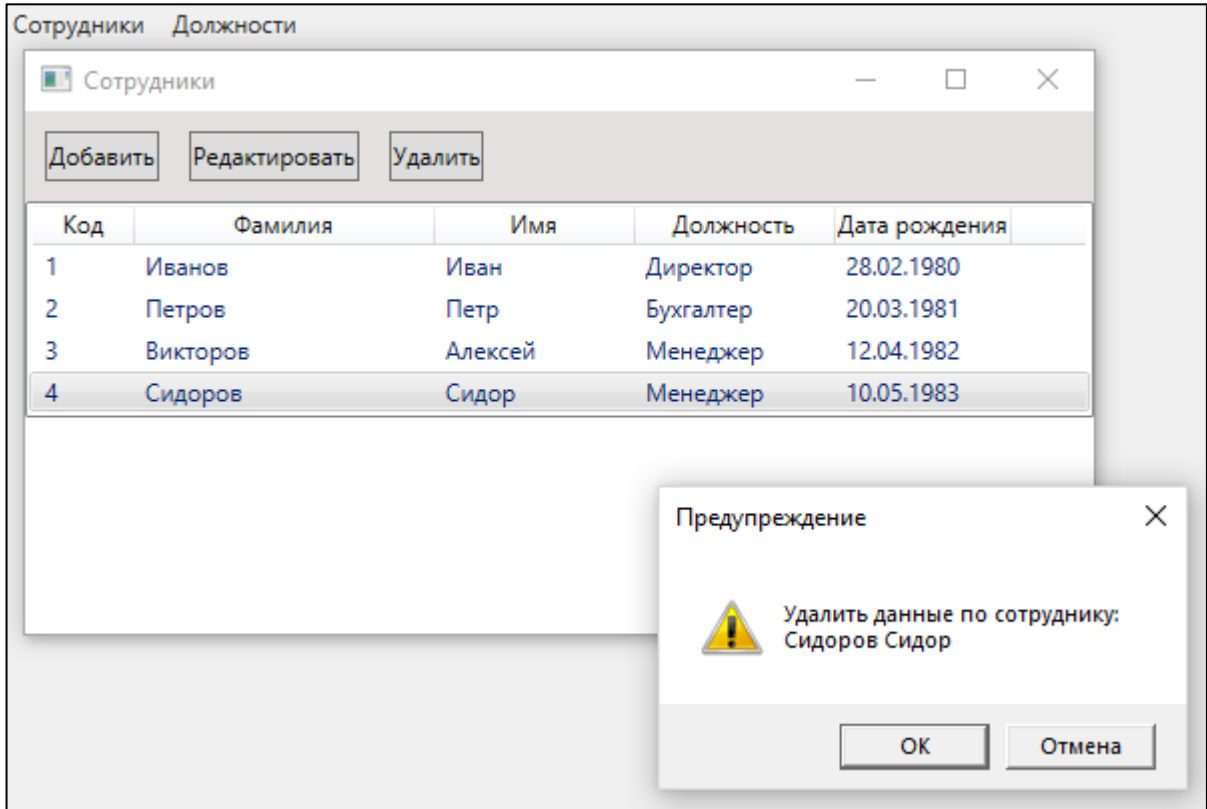


Рисунок 5.12 – Удаление данных по сотруднику

Применение стилей

Анализ кода приложения показывает, что в двух окнах `WindowsEmployee` и `WindowsRole` для выполнения действий с данными используются кнопки. Общее количество кнопок 6. Для каждой кнопки задается некоторые параметры. С целью обеспечения однотипности представления кнопок, близких по функциональности, применим для них пользовательский стиль.

Стиль будем создавать в словаре ресурсов. Для этого создадим новый объект в приложении `Dictionary.xml` на основе шаблона Словарь ресурсов (рисунок 5.13).

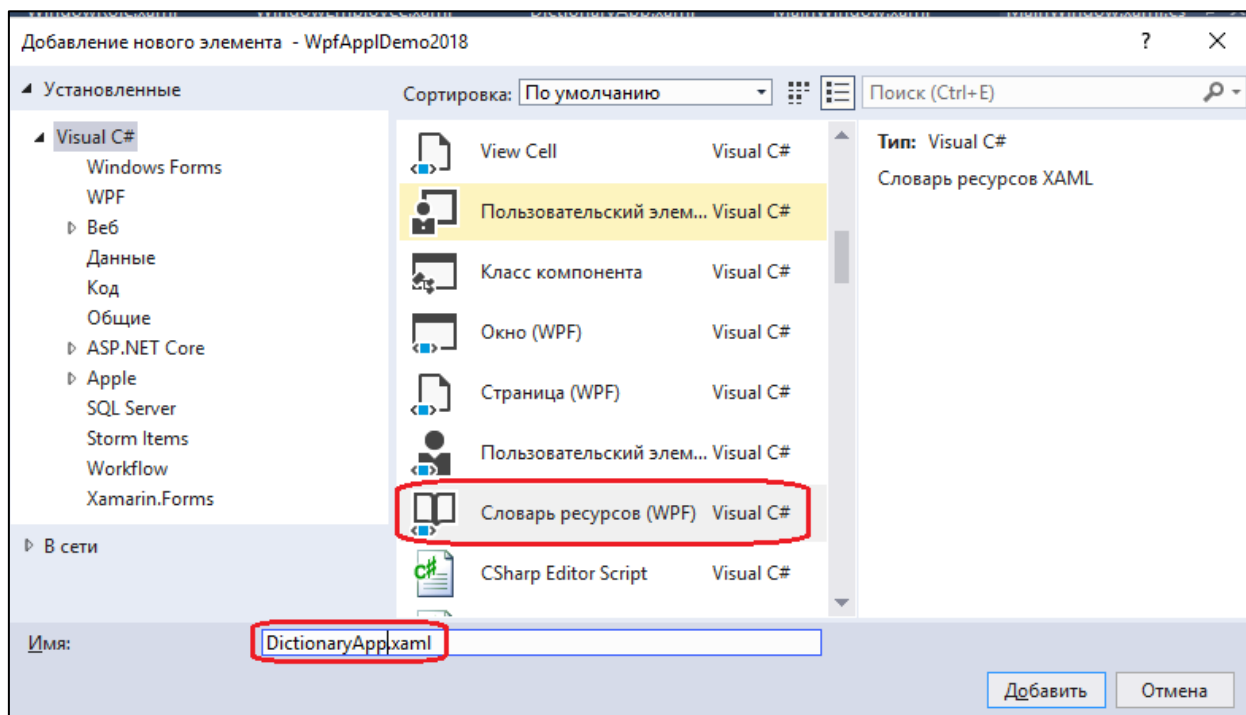


Рисунок 5.13 – Создание словаря ресурсов

Словарь ресурсов представляет собой xml-файл, в который можно поместить описание стилей и шаблонов. Для кнопок приложения сформируем стиль с ключем `ButtonMenu`, в котором определим такие свойства кнопок как высота (`Height`), внешние (`Margin`) и внутренние (`Padding`) отступы и толщину рамки (`BorderThickness`).

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:WpfApplDemo2018">
    <Style x:Key="ButtonMenu">
        <Setter Property="Control.Height" Value="25"/>
        <Setter Property="Control.Margin" Value="10,10,5,10"/>
        <Setter Property="Control.Padding" Value="5,0,5,0"/>
        <Setter Property="Control.BorderThickness" Value="2"/>
    </Style>
</ResourceDictionary>
```

Подключение словаря ресурсов на уровне приложения производится в XAML-коде класса `Application`.

```
<Application x:Class="WpfApplDemo2018.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:WpfApplDemo2018"
    StartupUri="MainWindow.xaml">
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="DictionaryApp.xaml" />
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
```

```

    </ResourceDictionary>
</Application.Resources>
</Application>

```

При таком способе объявления ресурсы, включенные в словарь DictionaryApp.xaml будут доступны во всем приложении.

Внесем изменения в XAML-код окна WindowRole в части описания кнопок.

```

<StackPanel Orientation="Horizontal" Background="#FFE4E0E0">
    <Button Style="{StaticResource ButtonMenu}" Content="Добавить"
        Command="{Binding AddRole }"/>
    <Button Style="{StaticResource ButtonMenu}" Content="Редактировать"
        Command="{Binding EditRole}"/>
    <Button Style="{StaticResource ButtonMenu}" Content="Удалить"
        Command="{Binding DeleteRole}"/>
</StackPanel>

```

Стиль кнопок задается с помощью свойства Style как статический ресурс:

```
Style="{StaticResource ButtonMenu}"
```

Пересоберем приложение и запустим отладку. По пункту меню Должности перейдем в окно Должности (рисунок 5.14).

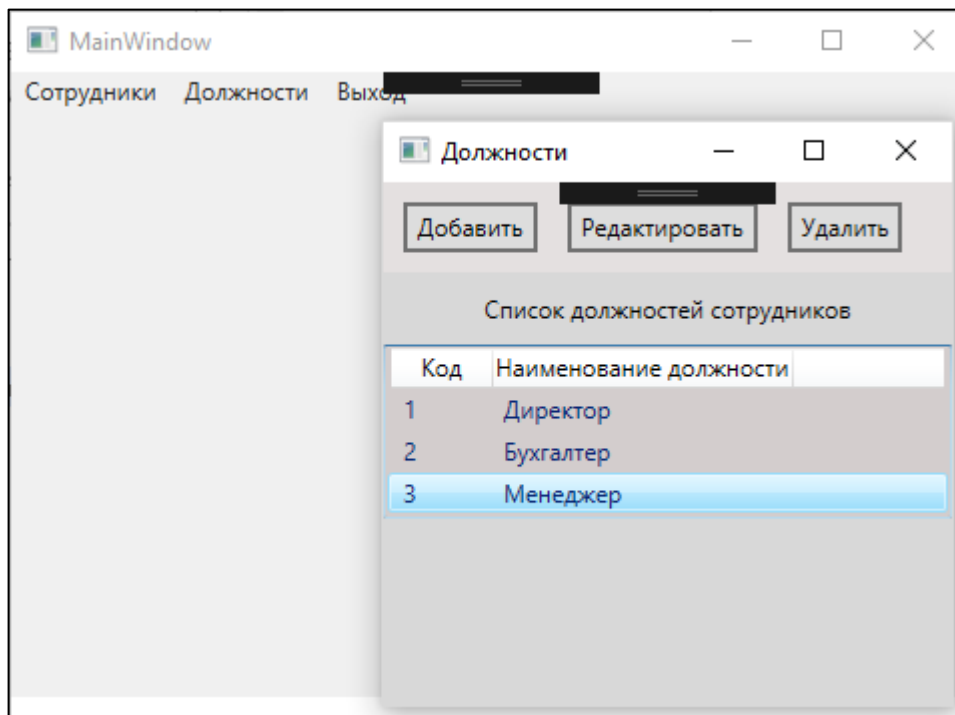


Рисунок 5.14 – Создание словаря ресурсов

Отображение кнопок на рисунке 5.14 соответствует определенному нами стилю.

Аналогичные изменения в описании кнопок необходимо сделать для окна WindowEmployee.

Задание на лабораторную работу

1. Сделайте копию проекта, созданного в лабораторной работе 6.
2. Создайте модель команд и примените её для вашего приложения в соответствии с методикой изложенной в описании данной лабораторной работы.
3. Создайте словарь ресурсов и используйте его в приложении.
4. Протестировать добавленную функциональность.

ЛАБОРАТОРНАЯ РАБОТА 6. Разработка приложений, взаимодействующего с данными в формате json

Цель работы: Получить навыки разработки приложений, взаимодействующих с данными, представленными в формате json.

Общие сведения о формате данных JSON

JSON (JavaScript Object Notation) - текстовый формат обмена данными. Он основан на подмножестве языка программирования JavaScript. JSON - формат, полностью независимый от языка реализации, он использует в таких языках как C, C++, C#, Java, JavaScript, Perl, Python и многих других для обмена данными.

JSON основан на двух структурах данных:

- коллекция пар ключ/значение. В разных языках, эта концепция реализована как *объект*, запись, структура, словарь, хэш, именованный список или ассоциативный массив;
- упорядоченный список значений. В большинстве языков это реализовано как *массив*, вектор, список или последовательность.

Это универсальные структуры данных. Почти все современные языки программирования поддерживают их в какой-либо форме.

Формирование данных для приложения

Создадим копию проекта, созданного в лабораторной работе 7, в папке WpfApplDemo2018_Json. В дальнейшем будем модифицировать проект с целью использования данных в формате json.

Создадим папку DataModels для данных. В созданную папку добавим два файла RoleData.json и PersonData.json с данными по должностям сотрудников и данными по сотрудникам.

Файл RoleData.json.

```
[
  {
    "Id": 0,
    "NameRole": "Не задано"
  },
  {
    "Id": 1,
    "NameRole": "Директор"
  },
  {
    "Id": 2,
    "NameRole": "Бухгалтер"
  },
  {

```

Файл PersonData.json

```
[
  {
    "Id": 1,
    "RoleId": 1,
    "FirstName": "Иван",
    "LastName": "Иванов",
    "Birthday": "28.02.1980"
  },
  {
    "Id": 2,
    "RoleId": 2,
    "FirstName": "Петр",
    "LastName": "Петров",
    "Birthday": "20.03.1981"
  },
  {
    "Id": 3,
    "RoleId": 3,
    "FirstName": "Виктор",
    "LastName": "Викторов",
    "Birthday": "16.04.1982"
  },
  {
    "Id": 4,
    "RoleId": 3,
    "FirstName": "Сидор",
    "LastName": "Сидоров",
    "Birthday": "10.05.1983"
  },
]
```

С учетом того, что формат json – это текстовый формат целесообразно в модели данных изменить свойства классов на строковые. В нашем случае для класса Person.cs и PersonDpo.cs тип свойства Birthday изменим на string, т. е. public string Birthday { get; set; }

При изменении типа свойства Birthday необходимо внести изменения в конструкторах классов Person.cs и PersonDpo.cs.

Модификация функциональности обработки данных по должностям сотрудников

В классе RoleViewModel необходимо произвести изменения, касающиеся работы с данными по должностям.

Добавим поля поле path для определения места хранения json данных.

```
readonly string path = @"C:\Users\Alex\source\Project\WpfApplDemo2018\Json\WpfApplDemo2018\DataModels\ RoleData.json";
```

При обработке данных нам потребуются поля _jsonRoles и Error.

```
string _jsonRoles = String.Empty;
public string Error { get; set; }
```

Поле `_jsonRoles` необходимо для получения json файла с данными, а поле `Error` – для формирования сообщения при генерации ошибок.

Для загрузки данных из json файла будем использовать метод `LoadRole()`.

```
public ObservableCollection<Role> LoadRole()
{
    _jsonRoles = File.ReadAllText(path);
    if (_jsonRoles != null)
    {
        ListRole = JsonConvert.DeserializeObject<ObservableCollection<Role>>(_jsonRoles);
        return ListRole;
    }
    else
    {
        return null;
    }
}
```

В методе `LoadRole()` строка `_jsonRoles` с json файлом загружается из файла `RoleData.json` методом `File.ReadAllText(path)`.

```
_jsonRoles = File.ReadAllText(path);
```

После проверки корректности чтения файла `if (_jsonRoles != null)` вызывается метод десериализации `DeserializeObject` класса `JsonConvert` для формирования коллекции должностей `ListRole` из данных в json формате.

```
ListRole = JsonConvert.DeserializeObject<ObservableCollection<Role>>(_jsonRoles);
```

Метод `LoadRole()` возвращает коллекцию должностей.

Для сохранения измененных данных в json файле используем метод `SaveChanges(ObservableCollection<Role> listRole)`

```
private void SaveChanges(ObservableCollection<Role> listRole)
{
    var jsonRole = JsonConvert.SerializeObject(listRole);
    try
    {
        using (StreamWriter writer = File.CreateText(path))
        {
            writer.Write(jsonRole);
        }
    }
    catch (IOException e)
    {
        Error = "Ошибка записи json файла /n" + e.Message;
    }
}
```

В методе `SaveChanges` сериализуется коллекция `ListRole` в строку формата json.

```
var jsonRole = JsonConvert.SerializeObject(listRole);
```

Затем создается поток для записи текстовой информации

```
StreamWriter writer = File.CreateText(path)
```

и поток с помощью метода Write записывает текстовые данные в файл.

```
writer.Write(jsonRole);
```

При возникновении ошибки при записи формируется сообщение.

```
Error = "Ошибка записи json файла /n" + e.Message;
```

В методах AddRole, EditRole и DeleteRole необходимо добавить метод SaveChanges после обработки данных.

Полный код класса RoleViewModel.

```
using System;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.IO;
using System.Runtime.CompilerServices;
using System.Windows;
using Newtonsoft.Json;
using WpfApplDemo2018.Helper;
using WpfApplDemo2018.Model;
using WpfApplDemo2018.View;
namespace WpfApplDemo2018.ViewModel
{
    public class RoleViewModel: INotifyPropertyChanged
    {
        readonly string path = @"C:\Users\Alex\source\Project\WpfApplDemo2018\Json\WpfApplDemo2018\DataModels\RoleData.json";
        /// <summary>
        /// коллекция должностей сотрудников
        /// </summary>
        public ObservableCollection<Role> ListRole { get; set; } = new ObservableCollection<Role>();
        /// <summary>
        /// выбранная в списке должность
        /// </summary>
        private Role _selectedRole;
        /// <summary>
        /// выбранная в списке должность
        /// </summary>
        public Role SelectedRole
        {
            get
            {
                return _selectedRole;
            }
            set
            {
                _selectedRole = value;
                OnPropertyChanged("SelectedRole");
                EditRole.CanExecute(true);
            }
        }

        public string Error { get; set; }
    }
}
```

```

string _jsonRoles = String.Empty;
public RoleViewModel()
{
    ListRole = LoadRole();
}
#region command AddRole
/// команда добавления новой должности
private RelayCommand _addRole;
public RelayCommand AddRole
{
    get
    {
        return _addRole ??
            (_addRole = new RelayCommand(obj =>
            {
                WindowNewRole wnRole = new WindowNewRole
                {
                    Title = "Новая должность",
                };
                // формирование кода новой должности
                int maxIdRole = MaxId() + 1;
                Role role = new Role{Id = maxIdRole};
                wnRole.DataContext = role;
                if (wnRole.ShowDialog() == true)
                {
                    ListRole.Add(role);
                    SaveChanges(ListRole);
                }
                SelectedRole = role;
            },
            (obj) => true));
    }
}
#endregion
#region Command EditRole
/// команда редактирования должности
private RelayCommand _editRole;
public RelayCommand EditRole
{
    get
    {
        return _editRole ??
            (_editRole = new RelayCommand(obj =>
            {
                WindowNewRole wnRole = new WindowNewRole
                {
                    Title = "Редактирование должности",
                };
                Role role = SelectedRole;
                var tempRole = role.ShallowCopy();
                wnRole.DataContext = tempRole;
                if (wnRole.ShowDialog() == true)
            }
            ));
    }
}

```

```

        {
            // сохранение данных в оперативной памяти
            role.NameRole = tempRole.NameRole;
            SaveChanges(ListRole);
        }
    }, (obj) => SelectedRole != null && ListRole.Count >
0));
    }
}
#endregion
#region DeleteRole
/// команда удаления должности
private RelayCommand _deleteRole;
public RelayCommand DeleteRole
{
    get
    {
        return _deleteRole ??
            (_deleteRole = new RelayCommand(obj =>
            {
                Role role = SelectedRole;
                MessageBoxResult result = MessageBox.Show("Удалить
данные по должности: " + role.NameRole,
                    "Предупреждение", MessageBoxButton.OKCancel, Message-
BoxImage.Warning);
                if (result == MessageBoxResult.OK)
                {
                    ListRole.Remove(role);
                    SaveChanges(ListRole);
                }
            }, (obj) => SelectedRole != null && ListRole.Count > 0));
    }
}
#endregion
#region Methods
/// <summary>
/// загрузка json файла и десериализация данных для коллекции
должностей ListRole
/// </summary>
/// <returns></returns>
public ObservableCollection<Role> LoadRole()
{
    _jsonRoles = File.ReadAllText(path);
    if (_jsonRoles != null)
    {
        ListRole = JsonConvert.DeserializeObject<ObservableCollec-
tion<Role>>(_jsonRoles);
        return ListRole;
    }
    else
    {
        return null;
    }
}

```

```

    }
}
/// <summary>
/// Нахождение максимального Id в коллекции
/// </summary>
/// <returns></returns>
public int MaxId()
{
    int max = 0;
    foreach (var r in this.ListRole)
    {
        if (max < r.Id)
        {
            max = r.Id;
        };
    }
    return max;
}
/// <summary>
/// Сохранение json-строки с данными по должностям в файл
/// </summary>
/// <param name="listRole"></param>
private void SaveChanges(ObservableCollection<Role> listRole)
{
    var jsonRole = JsonConvert.SerializeObject(listRole);
    try
    {
        using (StreamWriter writer = File.CreateText(path))
        {
            writer.Write(jsonRole);
        }
    }
    catch (IOException e)
    {
        Error = "Ошибка записи json файла /n" + e.Message;
    }
}
#endregion
public event PropertyChangedEventHandler PropertyChanged;
protected virtual void OnPropertyChanged([CallerMemberName]
string propertyName = "")
{
    PropertyChanged?.Invoke(this, new PropertyChangedEven-
tArgs(propertyName));
}
}
}

```

После изменений, сделанных в классе RoleViewModel, изменения, вносимые в данные по должностям сотрудников (добавление, редактирование и удаление) будут сохраняться в файле RoleData.json.

Модификация функциональности обработки данных по сотрудникам

В классе `PersonViewModel` необходимо произвести изменения, касающиеся работы с данными по сотрудникам.

Добавим поля поле `path` для определения места хранения json данных.

```
readonly string path = @"C:\Users\Alex\source\Project\WpfApplDemo2018\Json\WpfApplDemo2018\DataModels\ Person.json";
```

При обработке данных нам потребуются поля `_jsonRoles` и `Error`.

```
string _jsonPersons = String.Empty;
public string Error { get; set; }
```

Поле `_jsonPersons` необходимо для получения json файла с данными, а поле `Error` – для формирования сообщения при генерации ошибок.

Для загрузки данных из json файла будем использовать метод `LoadPerson()`.

```
public ObservableCollection<Person> LoadPerson()
{
    _jsonPersons = File.ReadAllText(path);
    if (_jsonPersons != null)
    {
        ListPerson = JsonConvert.DeserializeObject<ObservableCollection<Person>>(_jsonPersons);
        return ListPerson;
    }
    else
    {
        return null;
    }
}
```

Метод `LoadPerson()` возвращает коллекцию данных по сотрудникам.

Для сохранения измененных данных в json файле используем метод `SaveChanges(ObservableCollection<Person> listRole)`

```
private void SaveChanges(ObservableCollection<Person> listPersons)
{
    var jsonPerson = JsonConvert.SerializeObject(listPersons);
    try
    {
        using (StreamWriter writer = File.CreateText(path))
        {
            writer.Write(jsonPerson);
        }
    }
    catch (IOException e)
    {
        Error = "Ошибка записи json файла /n" + e.Message;
    }
}
```

В методах AddPerson, EditPerson и DeletePerson необходимо добавить метод SaveChanges после обработки данных.

Полный код класса PersonViewModel.

```
using System;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.IO;
using System.Linq;
using System.Runtime.CompilerServices;
using System.Windows;
using Newtonsoft.Json;
using WpfApplDemo2018.Annotations;
using WpfApplDemo2018.Helper;
using WpfApplDemo2018.Model;
using WpfApplDemo2018.View;
namespace WpfApplDemo2018.ViewModel
{
    public class PersonViewModel: INotifyPropertyChanged
    {
        readonly string path = @"C:\Users\Alex\source\Project\WpfApplDemo2018\Json\WpfApplDemo2018\DataModels\PersonData.json";
        private PersonDpo _selectedPersonDpo;
        /// <summary>
        /// выделенные в списке данные по сотруднику
        /// </summary>
        public PersonDpo SelectedPersonDpo
        {
            get { return _selectedPersonDpo; }
            set
            {
                _selectedPersonDpo = value;
                OnPropertyChanged("SelectedPersonDpo");
            }
        }
        /// <summary>
        /// коллекция данных по сотрудникам
        /// </summary>
        public ObservableCollection<Person> ListPerson { get; set; }
        public ObservableCollection<PersonDpo> ListPersonDpo { get; set; }
        string _jsonPersons = String.Empty;
        public string Error { get; set; }
        public string Message { get; set; }
        public PersonViewModel()
        {
            ListPerson = new ObservableCollection<Person>();
            ListPersonDpo = new ObservableCollection<PersonDpo>();
            ListPerson = LoadPerson();
            ListPersonDpo = GetListPersonDpo();
        }
        #region AddPerson
        /// <summary>
```

```

/// добавление сотрудника
/// </summary>
private RelayCommand _addPerson;
/// <summary>
/// добавление сотрудника
/// </summary>
public RelayCommand AddPerson
{
    get
    {
        return _addPerson ??
            (_addPerson = new RelayCommand(obj =>
            {
                WindowNewEmployee wnPerson = new WindowNewEmployee
                {
                    Title = "Новый сотрудник"
                };
                // формирование кода нового сотрудника
                int maxIdPerson = MaxId() + 1;
                PersonDpo per = new PersonDpo
                {
                    Id = maxIdPerson,
                    Birthday = DateTime.Now.ToString(),
                };

                wnPerson.DataContext = per;
                if (wnPerson.ShowDialog() == true)
                {
                    var r = (Role)wnPerson.CbRole.SelectedValue;
                    if (r != null)
                    {
                        per.RoleName = r.NameRole;
                        per.Birthday = PersonDpo.GetStringBirth-
day(per.Birthday);
                        ListPersonDpo.Add(per);
                        // добавление нового сотрудника в коллекцию
ListRoles<Person>
                        Person p = new Person();
                        p = p.CopyFromPersonDpo(per);
                        ListPerson.Add(p);
                        try
                        {
                            // сохранение изменений в файле json
                            SaveChanges(ListPerson);
                        }
                        catch (Exception e)
                        {
                            Error = "Ошибка добавления данных в json файл\n" +
e.Message;
                        }
                    }
                }
            })
    }
}

```

```

        },
        (obj) => true));
    }
}
#endregion
#region EditPerson
/// команда редактирования данных по сотруднику
private RelayCommand _editPerson;
public RelayCommand EditPerson
{
    get
    {
        return _editPerson ??
            (_editPerson = new RelayCommand(obj =>
            {
                WindowNewEmployee wnPerson = new WindowNewEmployee()
                {
                    Title = "Редактирование данных сотрудника",
                };
                PersonDpo personDpo = SelectedPersonDpo;
                var tempPerson = personDpo.ShallowCopy();
                wnPerson.DataContext = tempPerson;

                if (wnPerson.ShowDialog() == true)
                {
                    // сохранение данных в оперативной памяти
                    // перенос данных из временного класса в класс отобра-
жения данных
                    var r = (Role)wnPerson.CbRole.SelectedValue;
                    if (r != null)
                    {
                        personDpo.RoleName = r.NameRole;
                        personDpo.FirstName = tempPerson.FirstName;
                        personDpo.LastName = tempPerson.LastName;
                        personDpo.Birthday = PersonDpo.GetStringBirth-
day(tempPerson.Birthday);
                        // перенос данных из класса отображения данных в
класс Person
                        var per= ListPerson.FirstOrDefault(p => p.Id == per-
sonDpo.Id);
                        if (per != null)
                        {
                            per = per.CopyFromPersonDpo(personDpo);
                        }
                        try
                        {
                            // сохранение данных в файле json
                            SaveChanges(ListPerson);
                        }
                        catch (Exception e)
                        {

```

```

        Error = "Ошибка редактирования данных в json файл\n"
+ e.Message;
    }
    }
    else
    {
        Message = "Необходимо выбрать должность сотрудника.";
    }
}
}, (obj) => SelectedPersonDpo != null && ListPer-
sonDpo.Count > 0));
}
}
#endregion
#region DeletePerson
/// команда удаления данных по сотруднику
private RelayCommand _deletePerson;
public RelayCommand DeletePerson
{
    get
    {
        return _deletePerson ??
            (_deletePerson = new RelayCommand(obj =>
            {
                PersonDpo person = SelectedPersonDpo;
                MessageBoxResult result = MessageBox.Show("Удалить
данные по сотруднику: \n" +
                    person.LastName + " " + person.FirstName,
                    "Предупреждение", MessageBoxButton.OKCancel, Message-
BoxImage.Warning);
                if (result == MessageBoxResult.OK)
                {
                    try
                    {
                        // удаление данных в списке отображения данных
                        ListPersonDpo.Remove(person);
                        // поиск удаляемого класса в коллекции ListRoles
                        var per = ListPerson.FirstOrDefault(p => p.Id == per-
son.Id);
                        if (per != null)
                        {
                            ListPerson.Remove(per);
                            // сохранение данных в файле json
                            SaveChanges(ListPerson);
                        }
                    }
                    catch (Exception e)
                    {
                        Error = "Ошибка удаления данных\n" + e.Message;
                    }
                }
            }
            )
            );
    }
}

```

```

        }, (obj) => SelectedPersonDpo != null && ListPersonDpo.Count > 0));
    }
}
#endregion
#region Method
/// <summary>
/// Загрузка данных по сотрудникам из json файла
/// </summary>
/// <returns></returns>
public ObservableCollection<Person> LoadPerson()
{
    _jsonPersons = File.ReadAllText(path);
    if (_jsonPersons != null)
    {
        ListPerson = JsonConvert.DeserializeObject<ObservableCollection<Person>>(_jsonPersons);
        return ListPerson;
    }
    else
    {
        return null;
    }
}
/// <summary>
/// Формирование коллекции классов PersonDpo из коллекции Person
/// </summary>
/// <returns></returns>
public ObservableCollection<PersonDpo> GetListPersonDpo()
{
    foreach (var person in ListPerson)
    {
        PersonDpo p = new PersonDpo();
        p = p.CopyFromPerson(person);
        ListPersonDpo.Add(p);
    }
    return ListPersonDpo;
}
/// <summary>
/// Нахождение максимального Id в коллекции данных
/// </summary>
/// <returns></returns>
public int MaxId()
{
    int max = 0;
    foreach (var r in this.ListPerson)
    {
        if (max < r.Id)
        {
            max = r.Id;
        }
    };
}

```

```

    }
    return max;
}
/// <summary>
/// Сохранение json-строки с данными по сотрудникам в json
файл
/// </summary>
/// <param name="listPersons"></param>
private void SaveChanges(ObservableCollection<Person> listPersons)
{
    var jsonPerson = JsonConvert.SerializeObject(listPersons);
    try
    {
        using (StreamWriter writer = File.CreateText(path))
        {
            writer.Write(jsonPerson);
        }
    }
    catch (IOException e)
    {
        Error = "Ошибка записи json файла /n" + e.Message;
    }
}
#endregion
public event PropertyChangedEventHandler PropertyChanged;
[NotifyPropertyChangedInvocator]
protected virtual void OnPropertyChanged([CallerMemberName]
string propertyName = "")
{
    PropertyChanged?.Invoke(this, new PropertyChangedEven-
tArgs(propertyName));
}
}
}

```

С учетом того, что тип свойства Birthday в модели данных изменено на string, процесс ввода данных по новым сотрудникам и изменение данных требует корректировки. При отображении даты рождения целесообразно представлять дату в текстовом поле, а при вводе или редактировании даты желательно использовать календарь. Для реализации такой функциональности внесем изменения в файл WindowNewEmployee.xaml в части XAML-кода для ввода даты рождения сотрудника. При текстовом отображении даты будем использовать элемент TextBox, а при отображении календаря – DatePicker. Причем оба элемента управления будут располагаться в одной и той же ячейке контейнера Grid и иметь идентичные параметны ширины и высоты.

```

<TextBox x:Name="tbBirthday" Style="{StaticResource Text-
BoxDate}"
    Text="{Binding Birthday}"
    Grid.Row="4" Grid.Column="1" Height="20"
    HorizontalAlignment="Left" VerticalAlignment="Center"

```

```

        Margin="5,8,0,7" Width="130"
        IsVisibleChanged="tbBirthday_IsVisibleChanged" />
<DatePicker x:Name="ClBirthday" Visibility="Hidden"
        Grid.Row="4" Grid.Column="1" SelectedDate="{Binding
        Birthday,
        StringFormat={{0:dd\}.{0:MM\}.{0:yyyy}},
        Mode=TwoWay,UpdateSourceTrigger=PropertyChanged}"
        HorizontalAlignment="Left" VerticalAlignment="Center"
        Margin="5,8,0,7" Width="130" />

```

Видимость текстового элемента `tbBirthday` определяется стилем `TextBoxDate`, который добавлен в библиотеку стилей.

```

<Style x:Key="TextBoxDate" TargetType="TextBox">
    <Style.Triggers>
        <Trigger Property="IsFocused" Value="True">
            <Setter Property="Visibility" Value="Hidden"/>
        </Trigger>
    </Style.Triggers>
</Style>

```

При отображении данных по сотруднику дата рождения отображается элементом `tbBirthday` в текстовом виде. Элемент `tbBirthday` становится невидимым (`Property="Visibility" Value="Hidden"`), когда на него устанавливается фокус (`Property="IsFocused" Value="True"`). Если фокус перемещается с элемента `tbBirthday`, то он вновь становится видимым.

Видимость календаря `ClBirthday` управляется методом `tbBirthday_IsVisibleChanged` при генерации события изменения видимости элемента `tbBirthday`.

```

private void tbBirthday_IsVisibleChanged(object sender, DependencyPropertyChangedEventArgs e)
{
    if (tbBirthday.Visibility == Visibility.Hidden)
    {
        ClBirthday.Visibility = Visibility.Visible;
    }
    else
    {
        ClBirthday.Visibility = Visibility.Hidden;
    }
}

```

При отображении данных по сотруднику окно `WindowNewEmployee.xaml` имеет вид, приведенный на рисунке 6.1, в котором Дата рождения отображается текстовым элементом `TextBox`/

Редактирование да...

Код: 5

Должность: Офис-менеджер

Фамилия: Полиева

Имя: Полина

Дата рождения: 23.02.1998

Сохранить Отменить

Рисунок 6.1 – Отображение данных по сотруднику

При редактировании даты рождения сотрудника курсор устанавливается на поле даты и окно WindowNewEmployee.xaml будет иметь вид, приведенный на рисунке 8.2, в котором Дата рождения отображается календарем – элемент DatePicker

Редактирование да...

Код: 5

Должность:

Фамилия: Полиева

Имя: Полина

Дата рождения: 23.02.1998

Март 1998

Пн	Вт	Ср	Чт	Пт	Сб	Вс
23	24	25	26	27	28	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

Рисунок 6.2 – Редактирование данных по сотруднику

Разработанное приложение обеспечивает обработку данных представленных в json формате.

Задание на лабораторную работу

1. Сделайте копию проекта, созданного в лабораторной работе 7.
2. Создайте json файлы для данных вашего приложения.
3. Модифицируйте ваше приложение в соответствии с методикой изложенной в описании данной лабораторной работы.
4. Протестировать изменения в функционировании приложения.

ЛАБОРАТОРНАЯ РАБОТА 7. Разработка приложений, взаимодействующего с моделью данных EDM

Цель работы: Получить навыки разработки приложений, взаимодействующих с данными, представленными моделью EDM.

Общие сведения о модели EDM

Платформа *Entity Framework* представляет собой набор технологий *ADO.NET*, обеспечивающих разработку приложений, связанных с обработкой данных. *Entity Framework* позволяет работать с данными в форме специфических для домена объектов и свойств, таких как клиенты и их адреса, без необходимости обращаться к базовым таблицам и столбцам базы данных, где хранятся эти данные. *Entity Framework* дает разработчикам возможность работать с данными на более высоком уровне абстракции; создавать и сопровождать приложения, ориентированные на данные, используя меньше кода, чем в традиционных приложениях. Поскольку *Entity Framework* является компонентом *.NET Framework*, приложения *Entity Framework* могут работать на любом компьютере, где установлена платформа *.NET Framework*, начиная с версии 3.5 с пакетом обновления 1 (*SP1*).

Платформа *Entity Framework* позволяет определить концептуальную модель, модель хранения и сопоставление между ними, которые оптимальным образом подходят для приложения. Средства модели EDM (*Entity Data Model*) в *Visual Studio* позволяют создать EDMX-файл из базы данных или графической модели, а затем обновлять его при их изменении.

Модель EDM — это набор основных понятий, которые описывают структуру данных независимо от формы хранения. Если данные имеют реляционную структуру, доступ к данным, хранение и масштабируемость будут весьма эффективными, однако написание эффективного и поддерживаемого кода становится более сложным.

Концептуальная модель — это специфическое представление структуры данных в виде сущностей и связей, которое обычно определяется на доменном языке *DSL*, реализующем основные понятия модели EDM. Язык определения концептуальной схемы (*CSDL*) — это пример такого доменного языка. Сущности и связи, описанные в концептуальной модели, можно представить в виде абстракций объектов и ассоциаций в приложении. Это позволяет разработчикам сфокусировать внимание на концептуальной модели, не думая о схеме хранения, и писать эффективный и поддерживаемый код. Одновременно разработчики схем хранения могут сфокусировать внимание на эффективности доступа к данным, хранения и масштабируемости.

Формирование данных для приложения

Создадим копию проекта, созданного в лабораторной работе 8, в папке *WpfApp1Demo2018_Ado*. В дальнейшем будем модифицировать проект с целью использования модели данных EDM.

В папке Model удалим класс PersonDpo. Внесем изменения в класс Role.

```
using System.Collections.Generic;
namespace WpfApplDemo2018.Model
{
    /// <summary>
    /// класс должность сотрудника
    /// </summary>
    public class Role
    {
        /// <summary>
        /// код должности
        /// </summary>
        public int Id { get; set; }
        /// <summary>
        /// наименование должности
        /// </summary>
        public string NameRole { get; set; }
        public Role()
        {
            this.Persons = new HashSet<Person>();
        }
        /// <summary>
        /// коллекция Persons для связи с классом Person
        /// </summary>
        public virtual ICollection<Person> Persons { get; set; }
    }
}
```

В классе Role оставим свойства Id и NameRole , описывающие данные по должности, и добавим коллекцию Persons, которая необходима для связи с классом Person:

```
public virtual ICollection<Person> Persons { get; set; }
```

Класс Person должен быть изменен следующим образом:

```
using System;
namespace WpfApplDemo2018.Model
{
    /// <summary>
    /// Класс сотрудник
    /// </summary>
    public class Person
    {
        /// <summary>
        /// код сотрудника
        /// </summary>
        public int Id { get; set; } // код
        /// <summary>
        /// код должности сотрудника
        /// </summary>
        public int RoleId { get; set; } // код должности
        /// <summary>
        /// имя сотрудника
        /// </summary>
    }
```

```

public string FirstName { get; set; } // имя
/// <summary>
/// фамилия сотрудника
/// </summary>
public string LastName { get; set; } // фамилия
/// <summary>
/// дата рождения сотрудника
/// </summary>
public DateTime Birthday { get; set; } // дата рождения
/// <summary>
/// класс должности для связи с сущностью Role
/// </summary>
public virtual Role Role { get; set; }
}

```

В классе Person оставим свойства, описывающие данные по сотруднику, и добавим свойство Role, которое необходимо для связи с классом Role:

```
public virtual Role Role { get; set; }
```

Для создания EDM модели добавим в проект новый элемент по шаблону Бодель ADO.NET EDM (1) с именем CompanyEntities (рисунок 7.1 - 2). При создании модели выберем технологию Code First, которая стоит модель с помощью кода (рисунок 7.2).

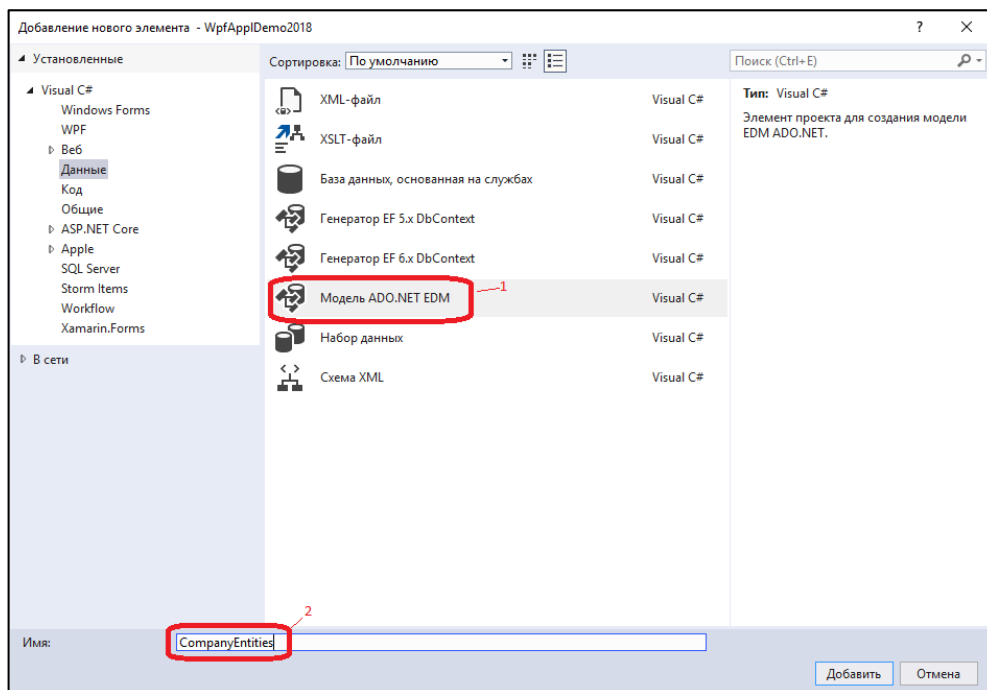


Рисунок 7.1 – Создание EDM-модели

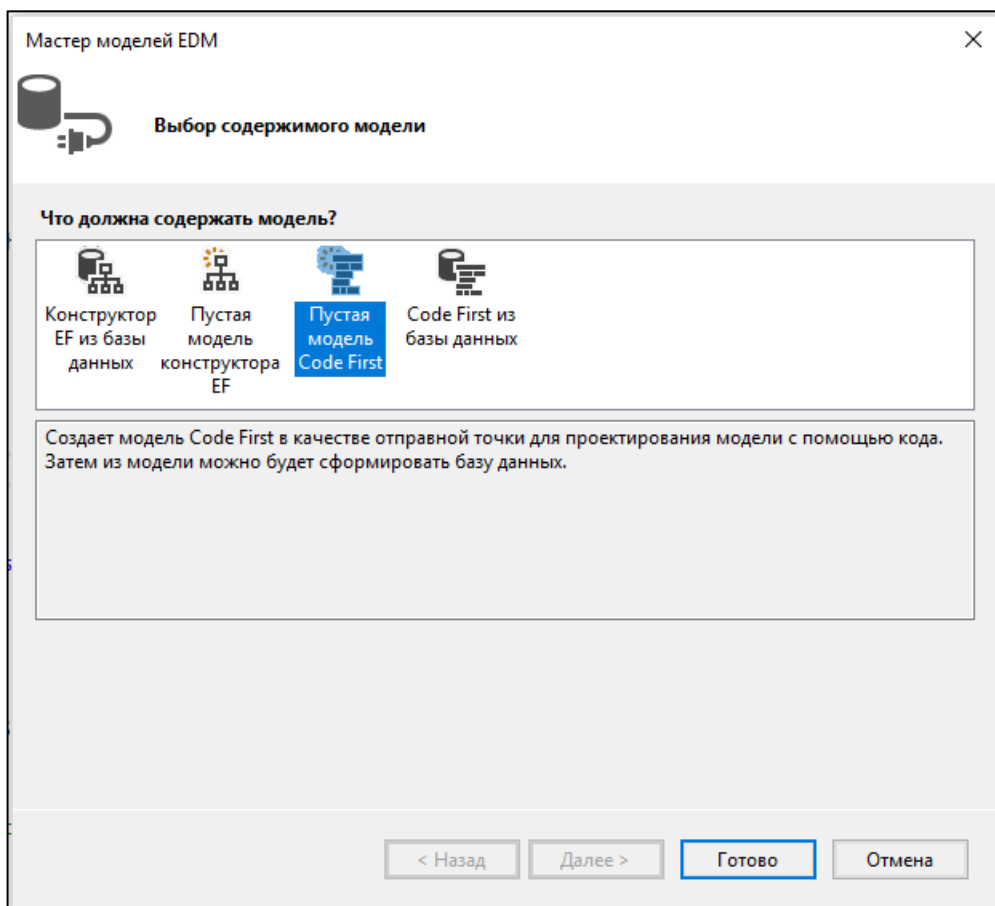


Рисунок 7.2 – Использование технологии Code First

При генерации модели будет сформирован класс `CompanyEntities.cs`, в который необходимо добавить сущности модели.

```
public virtual DbSet<Role> Roles{ get; set; }
public virtual DbSet<Person> Persons { get; set; }
```

Полный код класса `CompanyEntities`.

```
using WpfApplDemo2018.Model;
namespace WpfApplDemo2018
{
    using System.Data.Entity;
    public class CompanyEntities : DbContext
    {
        // Контекст настроен для использования строки подключения
        "CompanyEntities" из файла конфигурации
        // приложения (App.config или Web.config). По умолчанию эта
        строка подключения указывает на базу данных
        // "WpfApplDemo2018.CompanyEntities" в экземпляре LocalDb.
        //
        // Если требуется выбрать другую базу данных или поставщик
        базы данных, измените строку подключения "CompanyEntities"
        // в файле конфигурации приложения.
        public CompanyEntities()
            : base("name=CompanyEntities")
        {
        }
    }
}
```

```

// Добавьте DbSet для каждого типа сущности, который требуется
// включить в модель. Дополнительные сведения
// о настройке и использовании модели Code First см. в статье
http://go.microsoft.com/fwlink/?LinkId=390107.
public virtual DbSet<Role> Roles{ get; set; }
public virtual DbSet<Person> Persons { get; set; }
}
//public class MyEntity
//{
// public int Id { get; set; }
// public string Name { get; set; }
//}
}

```

В сгенерированном коде класса `CompanyEntities` имеется закомментированный пример класса `MyEntity`, который может использоваться для задания модели. В нашем случае нет необходимости создавать классы сущностей, так как мы уже их создали в папке `Model`. Это классы `Role` и `Person`.

После сборки приложения будет сгенерирован локальная база данных в SQL Server Express (рисунок 7.3)

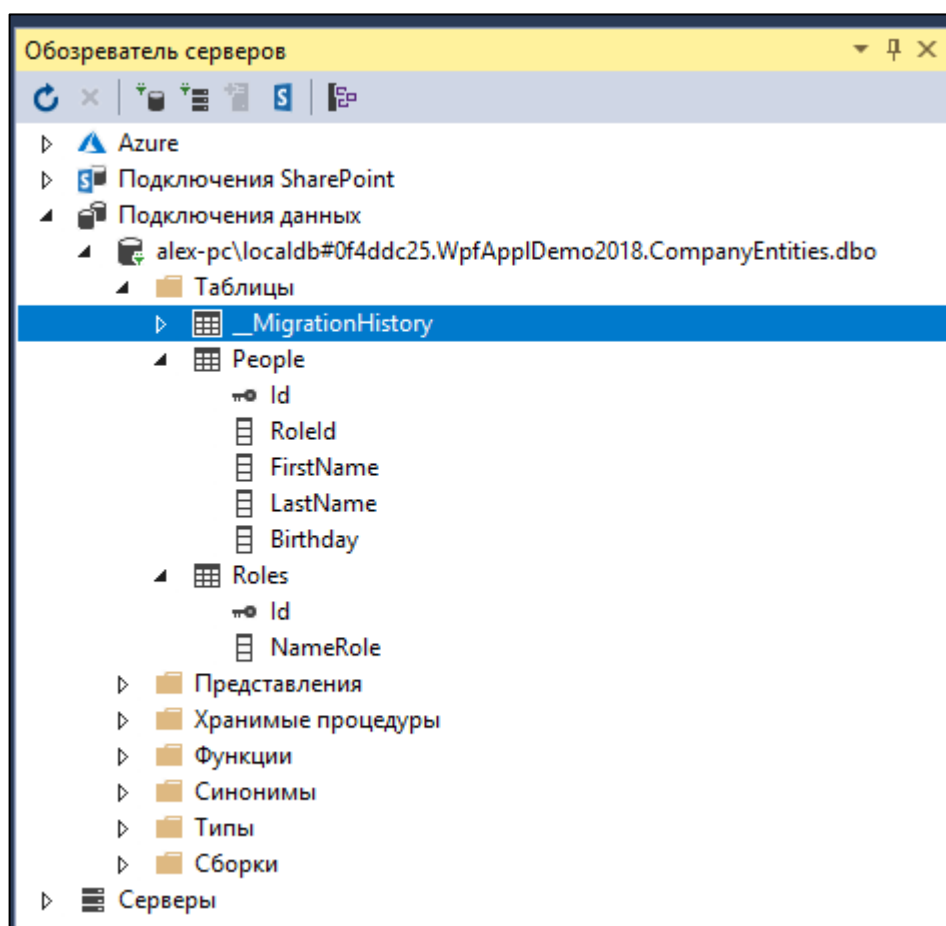


Рисунок 7.3 – Локальная база данных `CompanyEntities`

Описание таблиц `Role` и `People` в дизайнера Visual Studio приведено на рисунке 7.4 и 7.5.

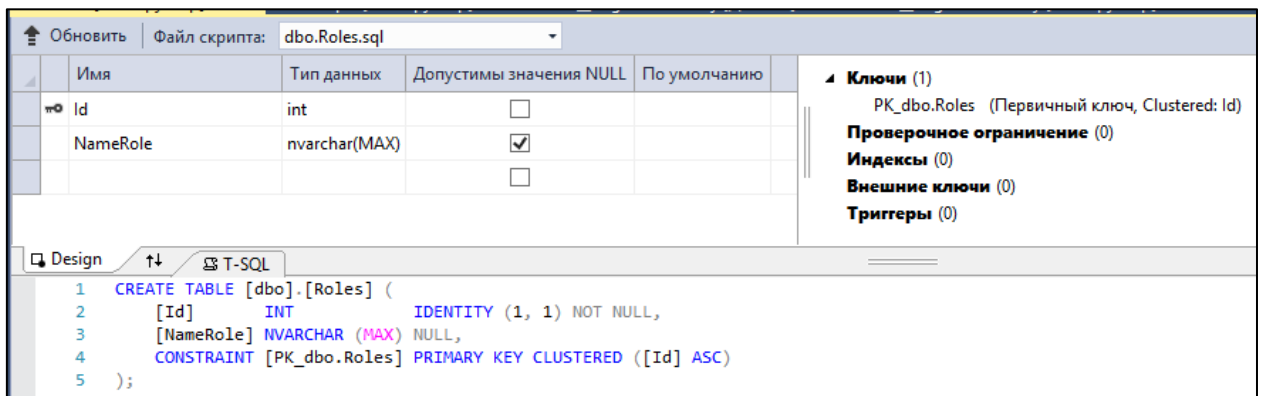


Рисунок 7.4 – Описание таблицы Role

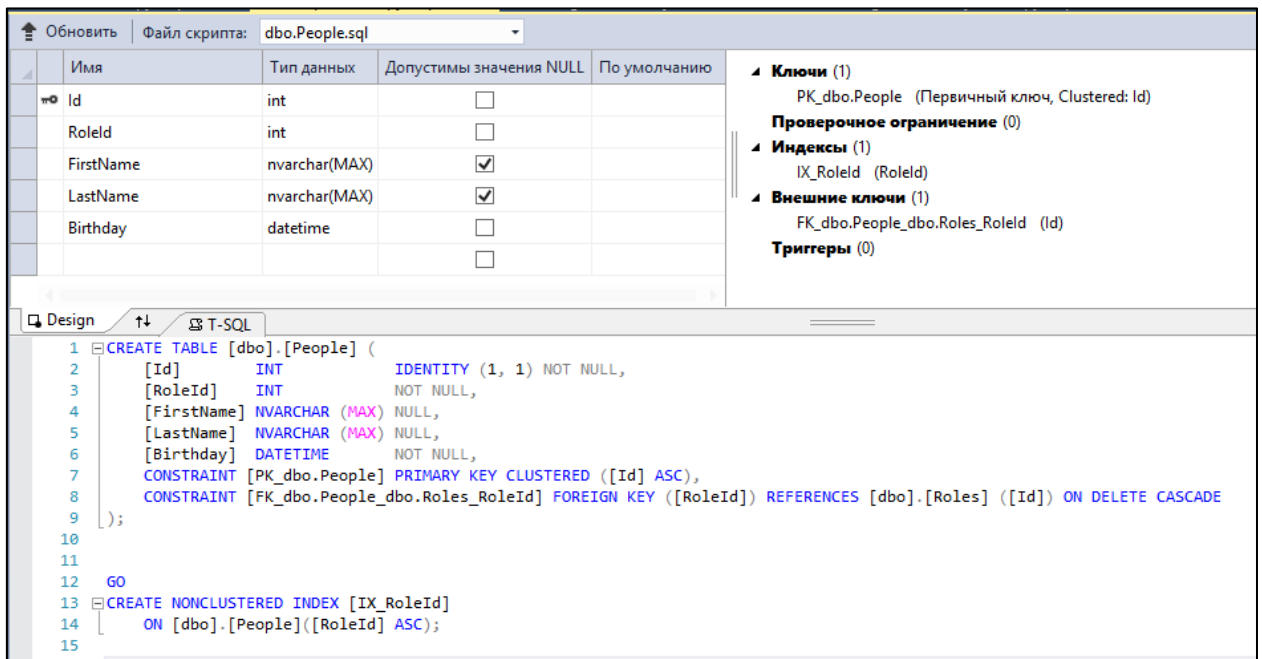


Рисунок 7.5 – Описание таблицы People

Загрузка, добавление, редактирование и удаление данных по должностям

Для обеспечения требуемой функциональности обработки данных по должностям сотрудников необходимо внести изменения в класс RoleViewModel.

Конструктор класса создает экземпляр коллекции ListRole и загружает данные по должностям сотрудников с помощью метода GetRoles().

```

public RoleViewModel()
{
    ListRole = new ObservableCollection<Role>();

    // Загрузка данных по должностям сотрудников
    ListRole = GetRoles();
}

```


Загрузка данных по должностям

Метод **GetRoles** создает контекст EDM модели context и с помощью LINQ запроса к базе данных формирует коллекцию **ListRole**.

```
private ObservableCollection<Role> GetRoles()
{
    using (var context = new CompanyEntities())
    {
        var query = from role in context.Roles
                     orderby role.NameRole
                     select role;
        if (query.Count() != 0)
        {
            foreach (var c in query)
                ListRole.Add(c);
        }
    }
    return ListRole;
}
```

Добавление новых данных по должностям

Метод **AddRole** добавляет новую должность сотрудника в базу данных.

Для добавления новой должности в контекст модели используется метод **Add**.

```
context.Roles.Add(newRole);
```

Сохранение данных в базе данных реализует метод SaveChanges().

```
context.SaveChanges();
```

Полный код команды AddRole.

```
#region command AddRole
/// команда добавления новой должности
private RelayCommand _addRole;
public RelayCommand AddRole
{
    get
    {
        return _addRole ??
            (_addRole = new RelayCommand(obj =>
            {
                Role newRole = new Role();
                WindowNewRole wnRole = new WindowNewRole
                {
                    Title = "Новая должность",
                    DataContext = newRole,
                };
                wnRole.ShowDialog();
                if (wnRole.DialogResult == true)
                {
                    using (var context = new CompanyEntities())
                    {
                        try
                        {
                            context.Roles.Add(newRole);
                        }
                    }
                }
            }
            ));
    }
}
```

```

        context.SaveChanges();
        ListRole.Clear();
        ListRole = GetRoles();
    }
    catch (Exception ex)
    {
        MessageBox.Show("\nОшибка добавления данных!\n" +
ex.Message, "Предупреждение");
    }
}
}, (obj) => true));
}
}
#endregion

```

Редактирование данных по должностям

Метод `EditRole` реализует редактирование данных по должности сотрудника. При редактировании должности в контексте находится экземпляр по `Id`

```
Role role = context.Roles.Find(editRole.Id);
```

Экземпляру `role` присваивается новое измененное значение.

```

if (role.NameRole != editRole.NameRole)
    role.NameRole = editRole.NameRole.Trim();

```

Измененное значение запоминается в базе данных.

```

try
{
    context.SaveChanges();
    ListRole.Clear();
    ListRole = GetRoles();
}

```

Полный код команды `EditRole`

```

#region Command EditRole
/// команда добавления новой должности
private RelayCommand _editRole;
public RelayCommand EditRole
{
    get
    {
        return _editRole ??
            (_editRole = new RelayCommand(obj =>
            {
                Role editRole = SelectedRole;
                WindowNewRole wnRole = new WindowNewRole
                {
                    Title = "Редактирование должности",
                    DataContext = editRole,
                };
                wnRole.ShowDialog();
                if (wnRole.DialogResult == true)
                {
                    using (var context = new CompanyEntities())

```

```

    {
        Role role = context.Roles.Find(editRole.Id);
        if (role.NameRole != editRole.NameRole)
            role.NameRole = editRole.NameRole.Trim();
        try
        {
            context.SaveChanges();
            ListRole.Clear();
            ListRole = GetRoles();
        }
        catch (Exception ex)
        {
            MessageBox.Show("\nОшибка редактирования данных!\n" +
ex.Message, "Предупреждение");
        }
    }
    else
    {
        ListRole.Clear();
        ListRole = GetRoles();
    }
}, (obj) => SelectedRole != null && ListRole.Count > 0));
}
}
#endregion

```

Удаление данных по должностям

Метод `DeleteRole` обеспечивает удаление данных по должности сотрудника из базы данных. Данные по удаляемой должности находятся в контексте по `Id`.

```
Role delRole = context.Roles.Find(role.Id);
```

Удаление найденного экземпляра реализуется методом `Remove`.

```
context.Roles.Remove(delRole);
```

Полный код команды `DeleteRole`.

```

#region DeleteRole
/// команда добавления новой должности
private RelayCommand _deleteRole;
public RelayCommand DeleteRole
{
    get
    {
        return _deleteRole ??
            (_deleteRole = new RelayCommand(obj =>
            {
                Role role = SelectedRole;
                using (var context = new CompanyEntities())
                {
                    // Поиск в контексте удаляемого автомобиля
                    Role delRole = context.Roles.Find(role.Id);
                    if (delRole != null)
                    {

```

```

        MessageBoxResult result = MessageBox.Show("Удалить
данные по должности: " + delRole.NameRole,
        "Предупреждение", MessageBoxButton.OKCancel, Mes-
sageBoxImage.Warning);
        if (result == MessageBoxResult.OK)
        {
            try
            {
                context.Roles.Remove(delRole);
                context.SaveChanges();
                ListRole.Remove(role);
            }
            catch (Exception ex)
            {
                MessageBox.Show("\nОшибка удаления данных!\n" +
ex.Message, "Предупреждение");
            }
        }
    }, (obj) => SelectedRole != null && ListRole.Count >
0));
    }
}
#endregion

```

Проверка обработки данных по должностям сотрудников

При выборе в главном меню пункта Должности выводится окно с данными по должностям (рисунок 7.6).

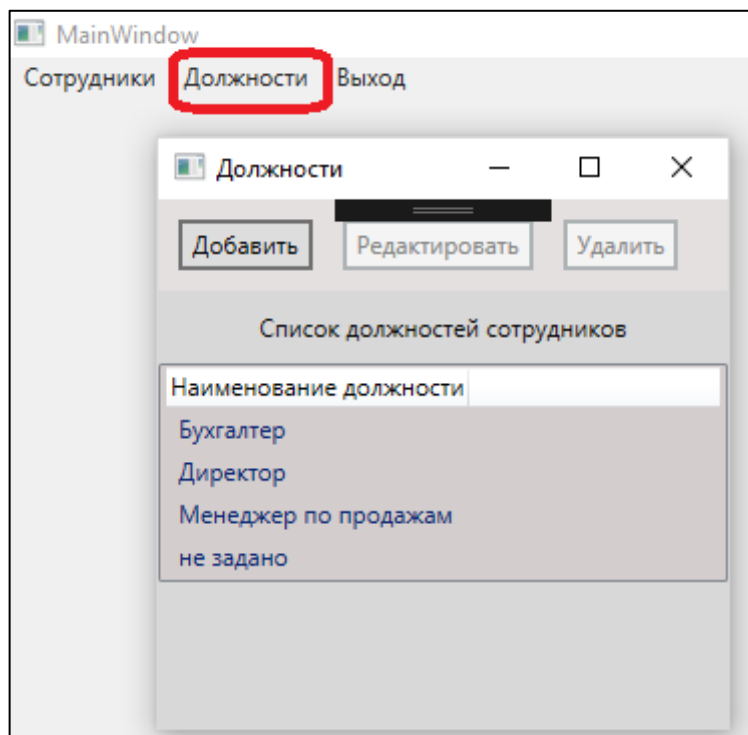


Рисунок 7.6 – Окно Список должностей

При нажатии на кнопку Добавить выводится окно Новая должность (рисунок 7.7).

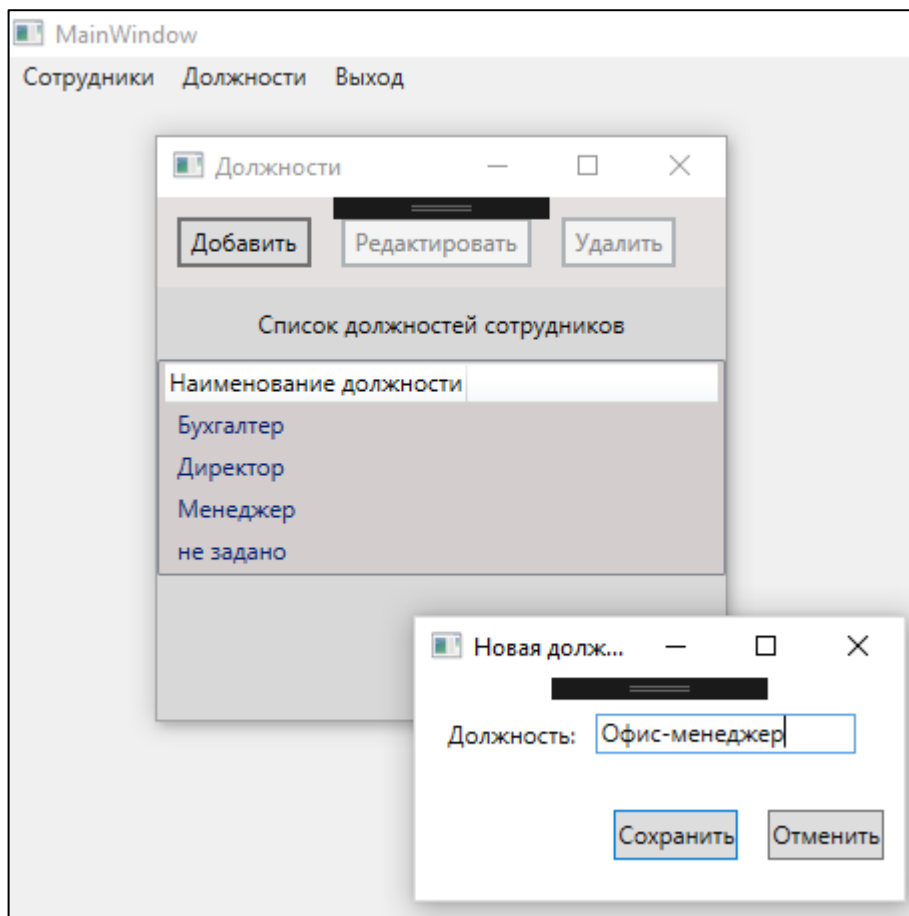


Рисунок 7.7 – Окно Новая должность

После добавления новой должности «Офис-менеджер» она появляется в списке должности (7.8).

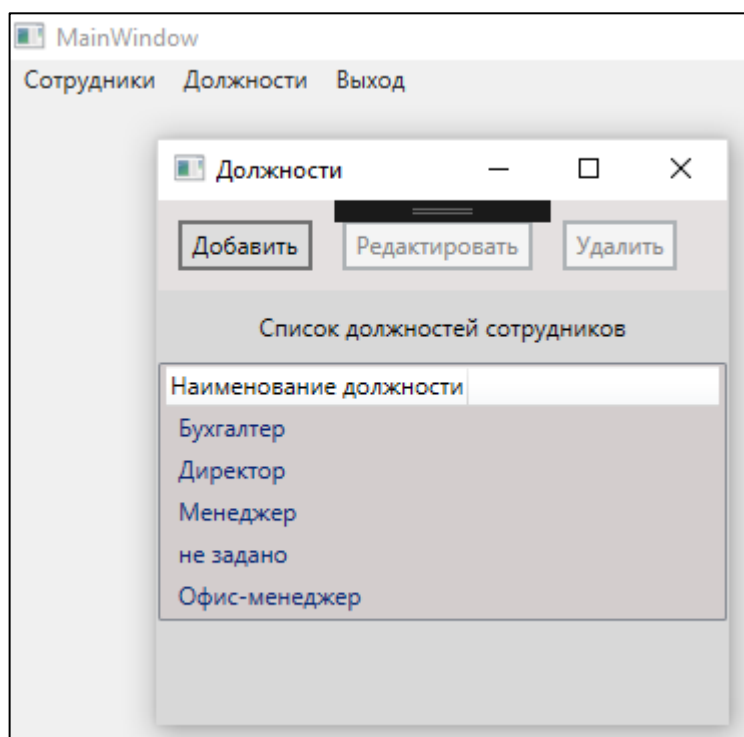


Рисунок 7.8 – Окно Список должностей с новой должностью
Для редактирования должности её необходимо выделить в списке должностей и нажать кнопку Редактировать (рисунок 7.9).

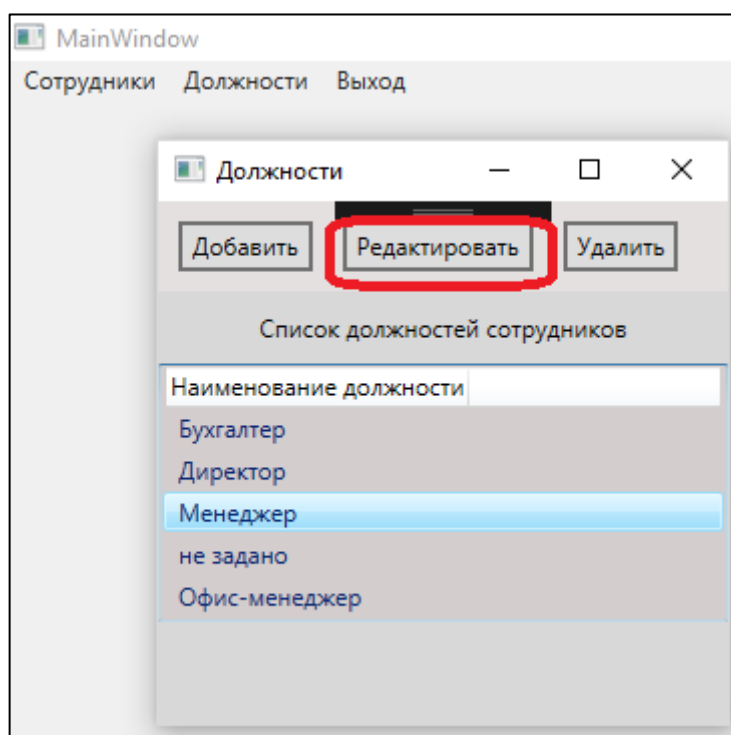


Рисунок 7.9 – Окно Список должностей – редактирование должности

Предположим, что необходимо изменить должность «Менеджер» на «Менеджер по продажам». Для этого в окне Должности выделяем строку с должность «Менеджер» и нажимаем кнопку Редактировать. В окне Редактирование должности вводим должность «Менеджет по продажам» (рисунок 7.10).

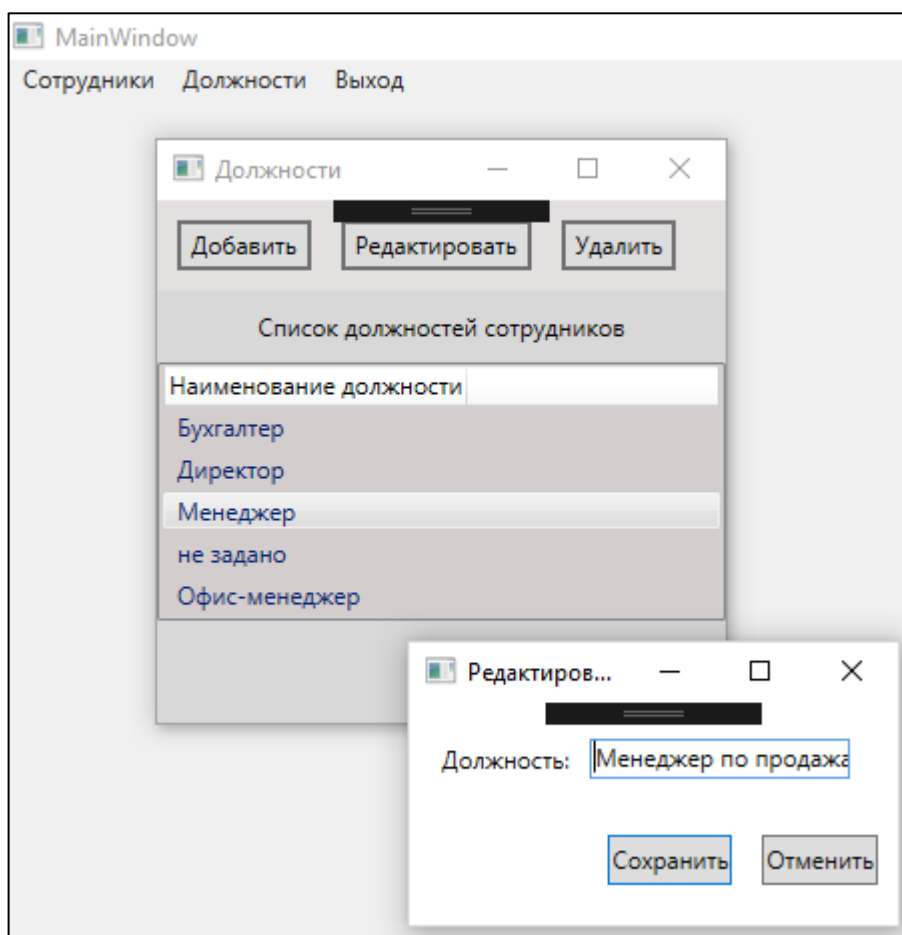


Рисунок 7.10 – Окно Редактирование должностей
Результат редактирования должности приведен на рисунке 7.11.

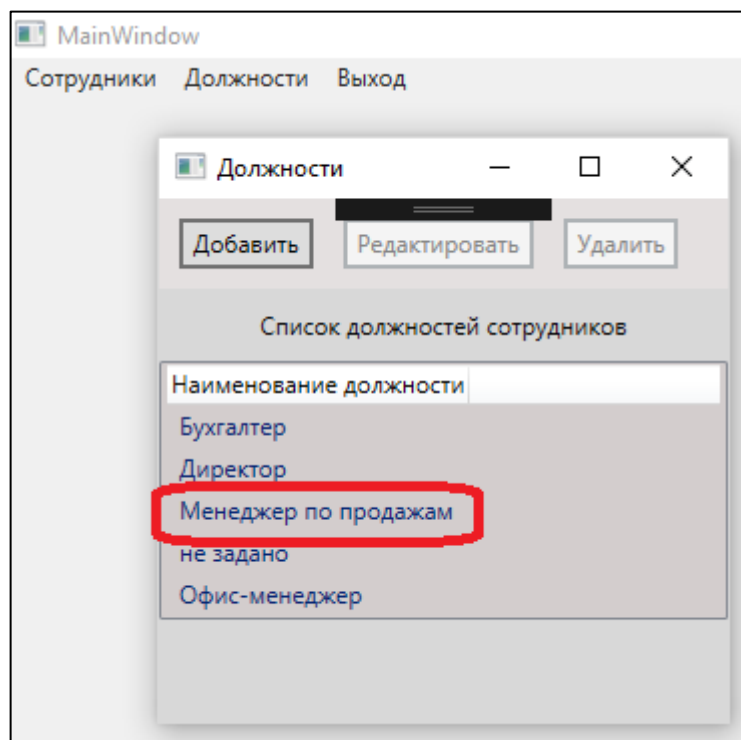


Рисунок 7.11 – Результат редактирования должности

Для удаления должности её необходимо выделить в списке должностей и нажать кнопку Удалить (рисунок 7.12).

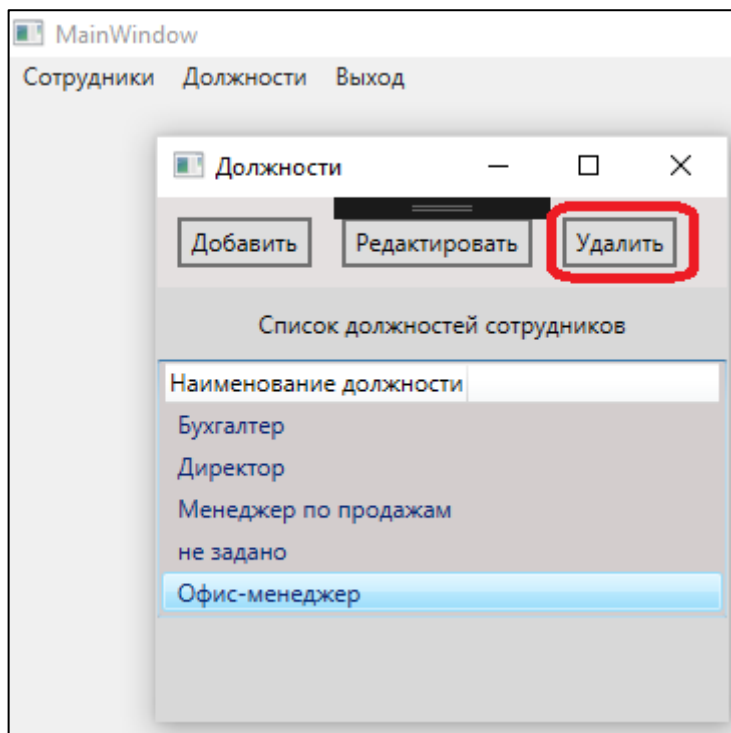


Рисунок 7.12 – Окно Список должностей – удаление должности

Предположим, что необходимо удалить должность «Офис-менеджер». Для этого в окне Должности выделяем строку с должностью «Офис-менеджер» и нажимаем кнопку Удалить. В результате должно быть выведено окно Предупреждение (рисунок 7.13).

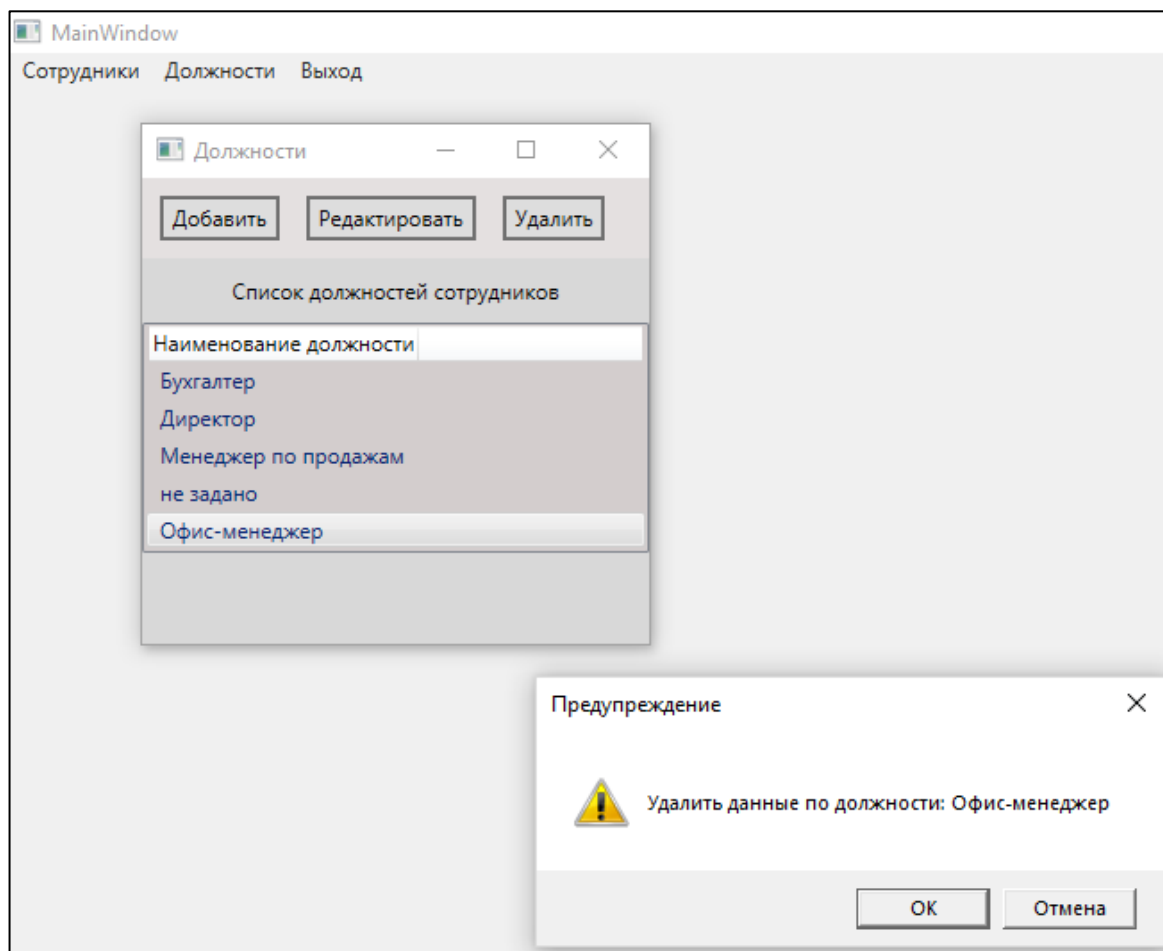


Рисунок 7.13 – Окно Предупреждения при удалении должности
Если подтвердить удаление, то должность «Офис-менеджер» будет удалена из базы данных приложения (рисунок 7.14).

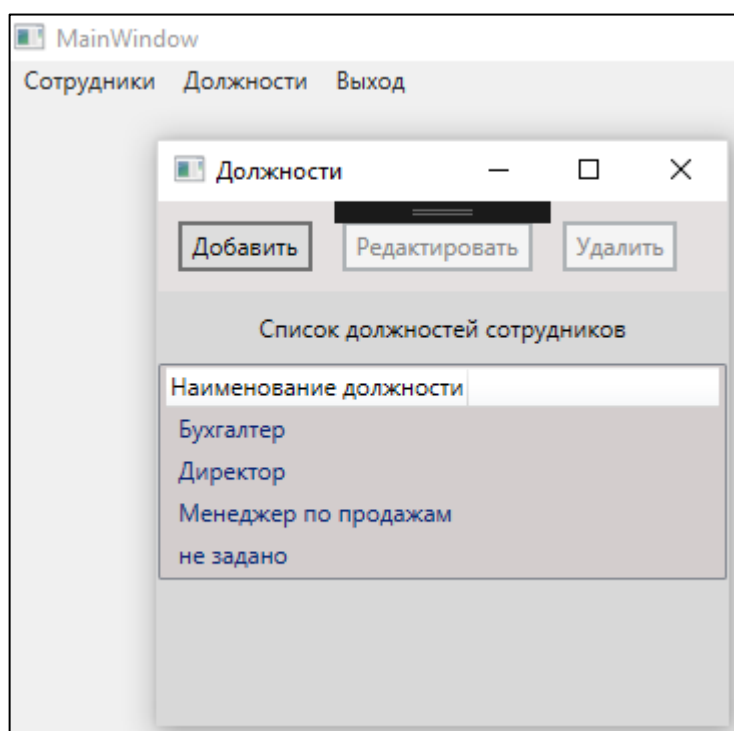


Рисунок 7.14 – Результат удаления должности

Загрузка, добавление, редактирование и удаление данных по сотрудникам

Для обеспечения требуемой функциональности обработки данных по сотрудникам необходимо внести изменения в класс `PersonViewModel`.

Загрузка данных по сотрудникам

Конструктор класса создает экземпляр коллекции `ListPerson` и загружает данные по должностям сотрудников с помощью метода `GetPersons()`.

```
public PersonViewModel()
{
    ListPerson = new ObservableCollection<Person>();
    ListPerson = GetPersons();
}
```

Метод `GetPersons` создает контекст EDM модели `context` и с помощью LINQ запроса к базе данных формирует коллекцию `ListPerson`.

```
private ObservableCollection<Person> GetPersons()
{
    using (var context = new CompanyEntities())
    {
        var query = from per in context.Persons
                    .Include("Role")
                    orderby per.LastName
                    select per;
        if (query.Count() != 0)
        {
            foreach (var p in query)
            {
                ListPerson.Add(p);
            }
        }
    }
    return ListPerson;
}
```

В Linq-запросе к сущности `Persons` подключается сущность `Role` с помощью расширяющего метода `Include("Role")`. Это позволяет получить доступ к должности сотрудника (`Role.NameRole`) через свойство `Role` класса `Person`.

```
var query = from per in context.Persons
            .Include("Role")
            orderby per.LastName
            select per;
```

В XAML коде окна `WindowEmployee.xaml` при привязке данных по должности сотрудника указывается свойство `Role.NameRole`, которое позволяет получить связанные данные из таблицы `Role`.

```
<GridViewColumn Header="Должность" Width="100"
    DisplayMemberBinding="{Binding Role.NameRole, Mode=TwoWay,
    UpdateSourceTrigger=PropertyChanged}"/>
```

В XAML коде окна `WindowNewEmployee.xaml` при привязке данных в `ComboBox` по должности сотрудника задаются свойства `DisplayMemberPath`,

SelectedValue и SelectedValuePath для корректного отображения данных из связанных сущностей.

```
<ComboBox x:Name="CbRole" Grid.Row="0" Grid.Column="1"
Height="20" Width="130"
HorizontalAlignment="Left" VerticalAlignment="Center" Mar-
gin="5"
ItemsSource="{Binding Source={StaticResource listRole}}"
DisplayMemberPath="NameRole"
SelectedValue="{Binding Path=RoleId, Mode=TwoWay, UpdateSource-
Trigger=PropertyChanged }"
SelectedValuePath="Id"/>
```

Добавление новых данных по сотруднику

Метод AddPerson добавляет данные по новому сотруднику в базу дан-
ных.

Для добавления нового сотрудника в контекст модели используется ме-
тод Add.

```
context.Persons.Add(newPerson);
```

Сохранение данных в базе данных реализует метод SaveChanges().

```
context.SaveChanges();
```

Полный код команды AddPerson.

```
#region AddPerson
/// <summary>
/// добавление сотрудника
/// </summary>
private RelayCommand _addPerson;
/// <summary>
/// добавление сотрудника
/// </summary>
public RelayCommand AddPerson
{
    get
    {
        return _addPerson ??
            (_addPerson = new RelayCommand(obj =>
            {
                Person newPerson = new Person
                {
                    Birthday = DateTime.Now
                };
                WindowNewEmployee wnPerson = new WindowNewEmployee
                {
                    Title = "Новый сотрудник",
                    DataContext = newPerson
                };
                wnPerson.ShowDialog();
                if (wnPerson.DialogResult == true)
                {
                    using (var context = new CompanyEntities())
                    {
                        try
```

```

        {
            Person ord = context.Persons.Add(newPerson);
            context.SaveChanges();
            ListPerson.Clear();
            ListPerson = GetPersons();
        }
        catch (Exception ex)
        {
            MessageBox.Show("\nОшибка добавления данных!\n" +
ex.Message, "Предупреждение");
        }
    }
    }, (obj) => true));
}
}
#endregion

```

Редактирование данных по сотруднику

Метод `EditPerson` реализует редактирование данных по сотруднику. При редактировании данных в контексте находится экземпляр по `Id`

```
Person person = context.Persons.Find(editPerson.Id);
```

Для экземпляра `person` присваиваются новые значения измененных свойств.

```

if (person.RoleId != editPerson.RoleId)
    person.RoleId = editPerson.RoleId;
if (person.FirstName != editPerson.FirstName)
    person.FirstName = editPerson.FirstName;
if (person.LastName != editPerson.LastName)
    person.LastName = editPerson.LastName;
if (person.Birthday != editPerson.Birthday)
    person.Birthday = editPerson.Birthday;

```

Измененное значение запоминается в базе данных.

```

try
{
    context.SaveChanges();
    ListPerson.Clear();
    ListPerson = GetPersons();
}

```

Полный код команды `EditPerson`

```

#region EditPerson
/// команда редактирования данных по сотруднику
private RelayCommand _editPerson;
public RelayCommand EditPerson
{
    get
    {
        return _editPerson ??
            (_editPerson = new RelayCommand(obj =>
            {
                Person editPerson = SelectedPerson;

```

```

WindowNewEmployee wnPerson = new WindowNewEmployee()
{
    Title = "Редактирование данных сотрудника",
    DataContext = editPerson
};
wnPerson.ShowDialog();
if (wnPerson.DialogResult == true)
{
    using (var context = new CompanyEntities())
    {
        Person person = context.Persons.Find(editPerson.Id);
        if (person != null)
        {
            if (person.RoleId != editPerson.RoleId)
                person.RoleId = editPerson.RoleId;
            if (person.FirstName != editPerson.FirstName)
                person.FirstName = editPerson.FirstName;
            if (person.LastName != editPerson.LastName)
                person.LastName = editPerson.LastName;
            if (person.Birthday != editPerson.Birthday)
                person.Birthday = editPerson.Birthday;
            try
            {
                context.SaveChanges();
                ListPerson.Clear();
                ListPerson = GetPersons();
            }
            catch (Exception ex)
            {
                MessageBox.Show("\nОшибка редактирования данных!\n"
+ ex.Message, "Предупреждение");
            }
        }
    }
}
else
{
    ListPerson.Clear();
    ListPerson = GetPersons();
}
}, (obj) => SelectedPerson != null && ListPerson.Count >
0));
}
}
#endregion

```

Удаление данных по сотруднику

Метод `DeletePerson` обеспечивает удаление данных по сотруднику из базы данных. Данные по удаляемому сотруднику находятся в контексте по `Id`.

```
Person person = context.Persons.Find(delPerson.Id);
```

Удаление найденного экземпляра реализуется методом `Remove`.

```
context.Persons.Remove(person);
```

Полный код команды DeletePerson.

```
#region DeletePerson
/// команда удаления данных по сотруднику
private RelayCommand _deletePerson;
public RelayCommand DeletePerson
{
    get
    {
        return _deletePerson ??
            (_deletePerson = new RelayCommand(obj =>
            {
                Person delPerson= SelectedPerson;
                using (var context = new CompanyEntities())
                {
                    // Поиск в контексте удаляемого автомобиля
                    Person person = context.Persons.Find(delPerson.Id);
                    if (person != null)
                    {
                        MessageBoxResult result = MessageBox.Show("Удалить
данные по сотруднику: \nФамилия: " + person.LastName +
                            "\nИмя: " + person.FirstName,
"Предупреждение", MessageBoxButton.OKCancel);
                        if (result == MessageBoxResult.OK)
                        {
                            try
                            {
                                context.Persons.Remove(person);
                                context.SaveChanges();
                                ListPerson.Remove(delPerson);
                            }
                            catch (Exception ex)
                            {
                                MessageBox.Show("\nОшибка удаления данных!\n" +
ex.Message, "Предупреждение");
                            }
                        }
                    }
                }
            }, (obj) => SelectedPerson != null && ListPerson.Count >
0));
    }
}

#endregion
```

Проверка обработки данных по сотрудникам

При выборе в главном меню пункта Сотрудники выводится окно с данными по сотрудникам (рисунок 7.15).

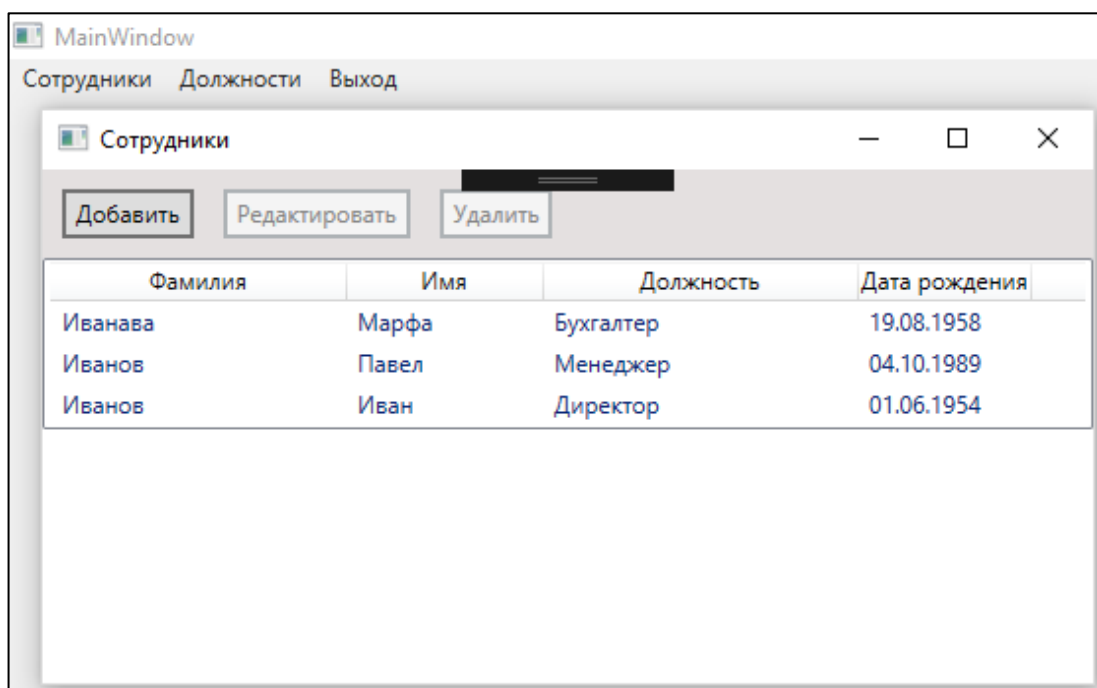


Рисунок 7.15 – Загрузка данных по сотрудникам

При добавлении данных по новому сотруднику необходимо в окне Сотрудники нажать кнопку Добавить и в окне Новый сотрудник ввести данные (рисунок 7.16).

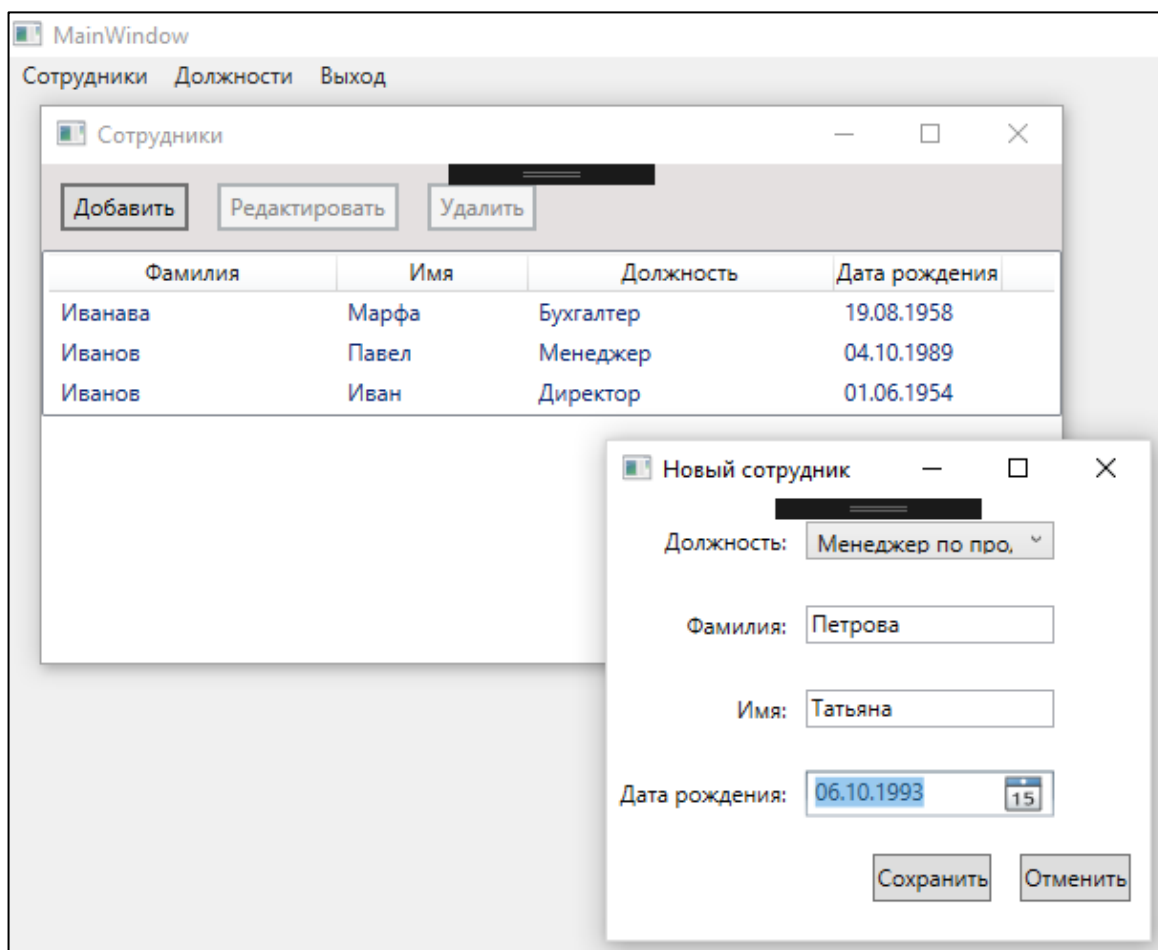


Рисунок 7.16 – Добавление данных по новому сотруднику

После подтверждения ввода данных по новому сотруднику его данные добавляются в базу данных и появляются в списке сотрудников (рисунок 7.17).

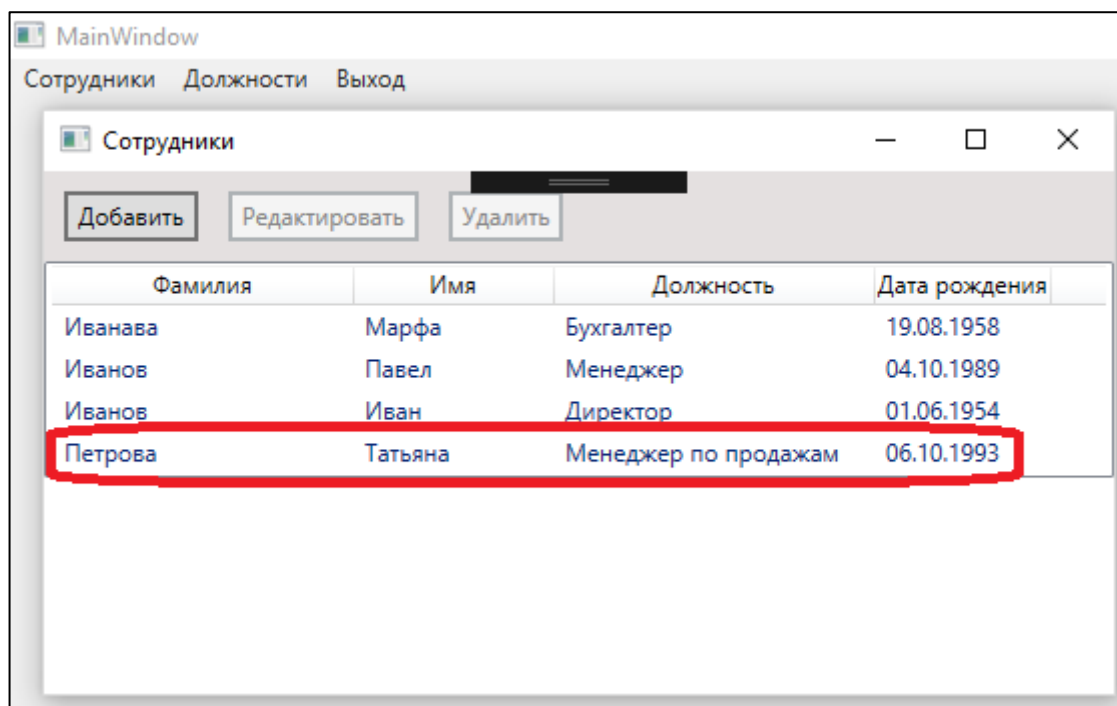


Рисунок 7.17 – Обновленные данные по сотрудникам

Для редактирования данных по сотруднику необходимо выделить строку в списке сотрудников и нажать кнопку Редактировать (рисунок 7.18).

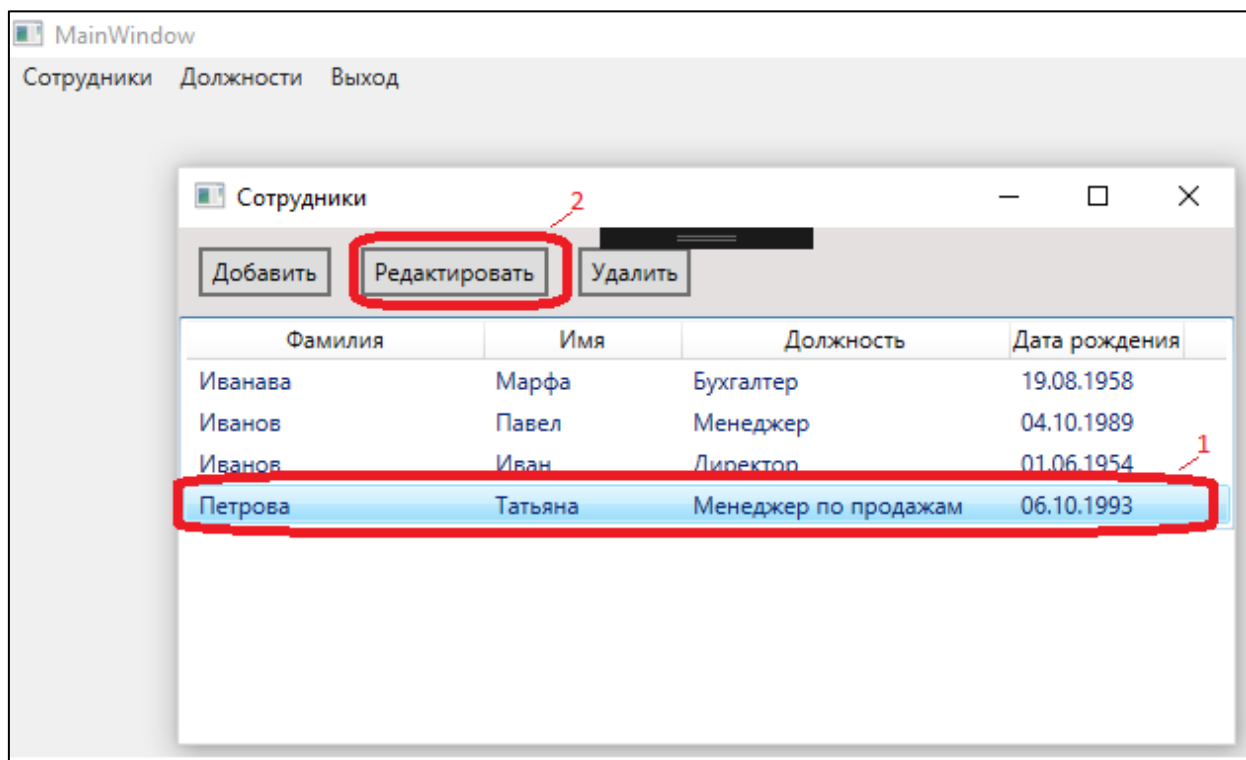


Рисунок 7.18 – Выбор данных для редактирования

В окне Редактирование данных необходимо изменить необходимые данные по сотруднику. В примере редактируется имя сотрудника (рисунок 7.19)

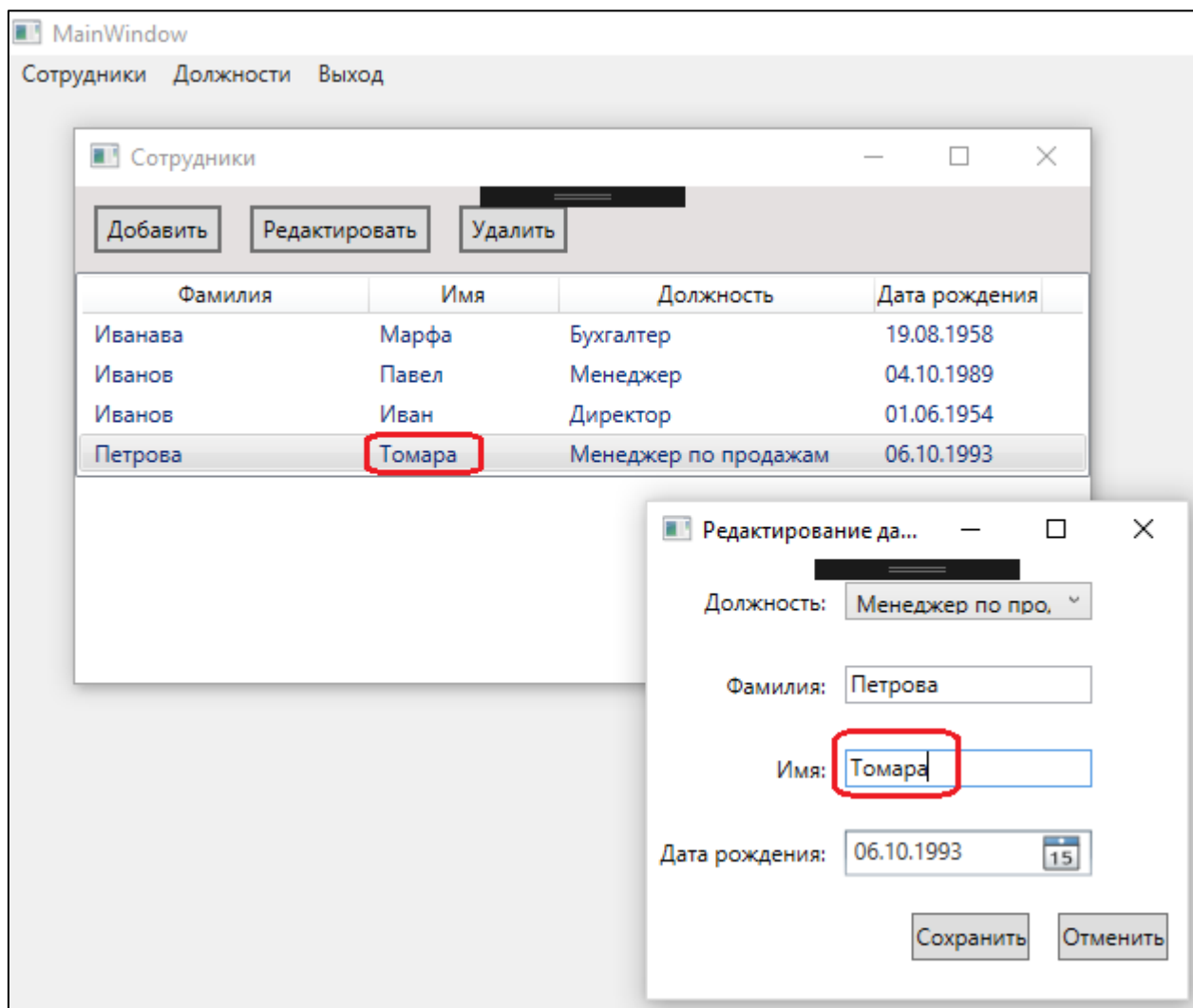


Рисунок 7.19 – Редактирование данных по сотруднику

После подтверждения редактирования данных по сотруднику его данные изменяются в базе данных и модифицируются в списке сотрудников (рисунок 7.20).

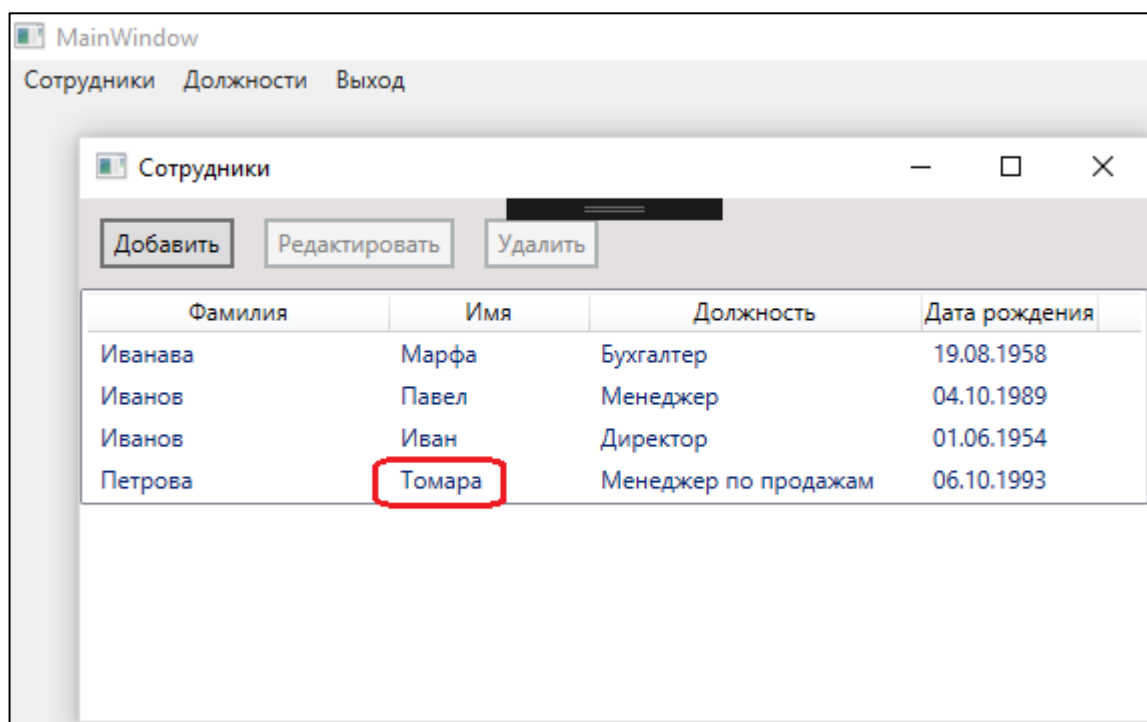


Рисунок 7.20 – Обновленные данные после редактирования

Для удаления данных по сотруднику необходимо выделить строку в списке сотрудников и нажать кнопку Удалить (рисунок 7.21).

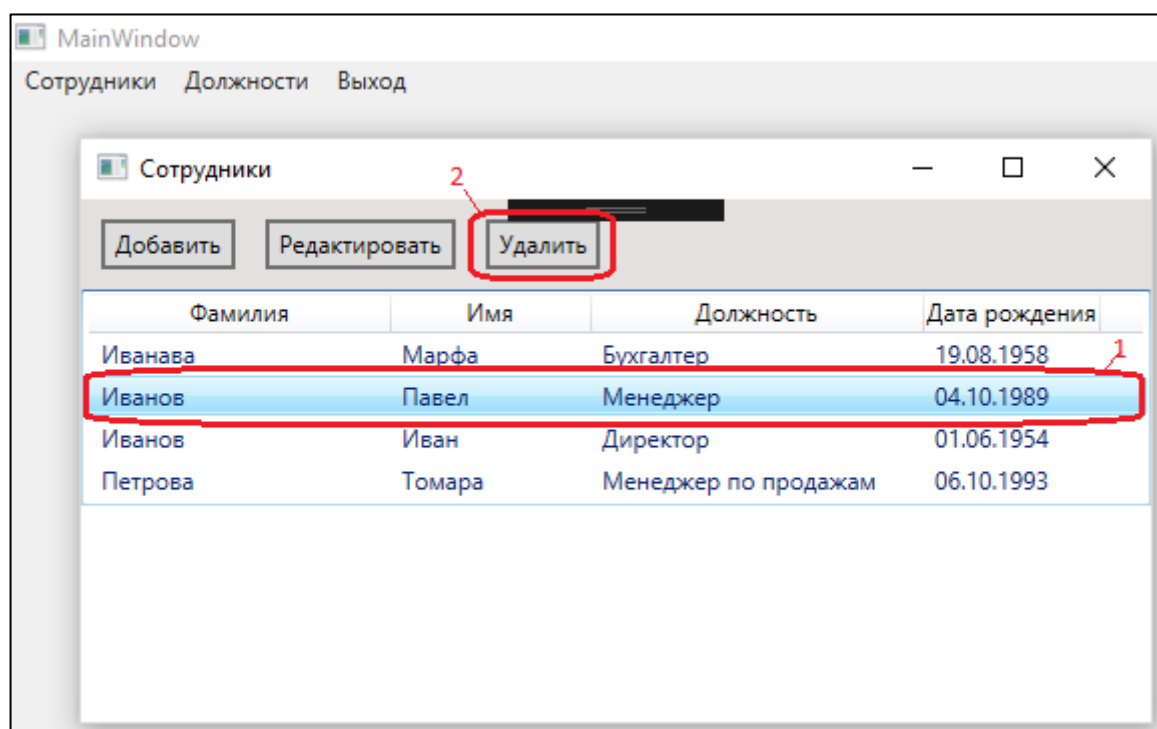


Рисунок 7.21 – Выбор данных для удаления

При удалении данных выводится предупреждение (рисунок 7.22).

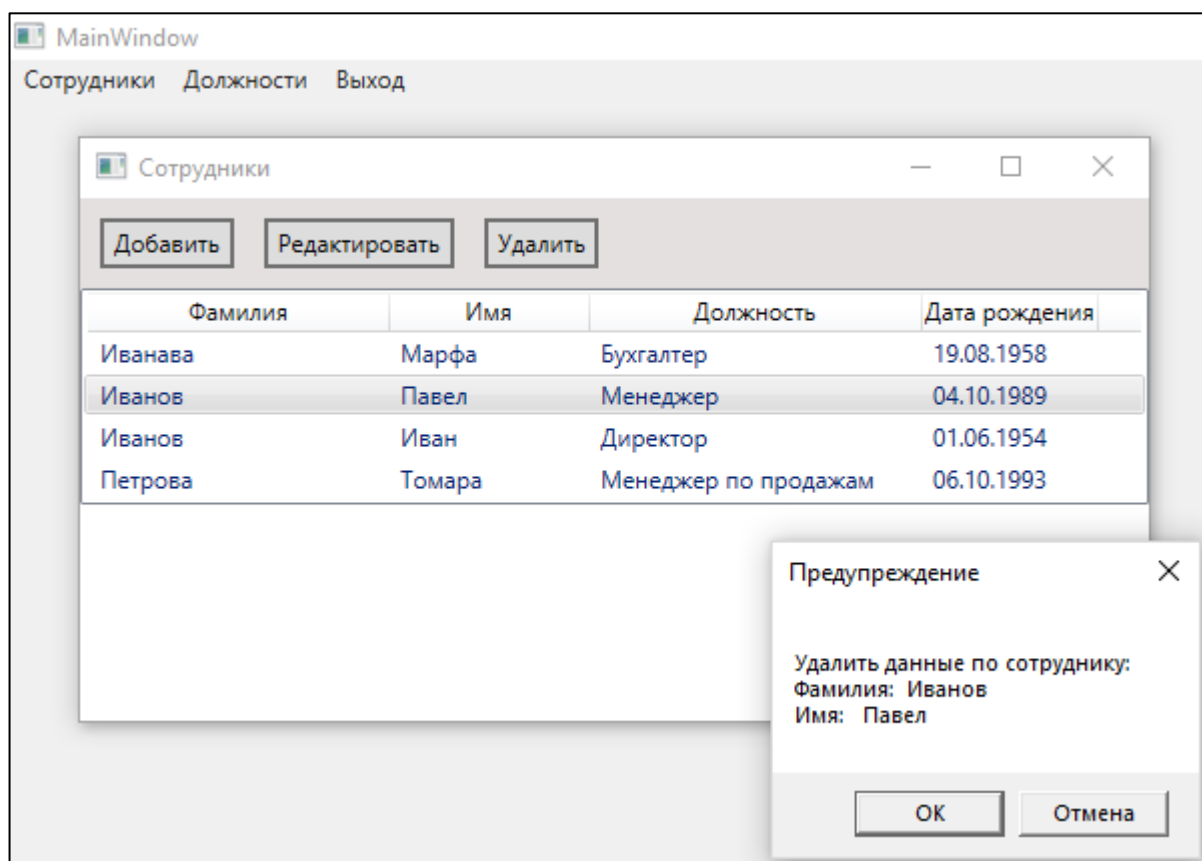


Рисунок 7.22 – Предупреждение об удалении данных

Если пользователь подтверждает удаление данных, то данные по сотруднику удаляются из базы данных и удаляются из списка сотрудников (7.23)

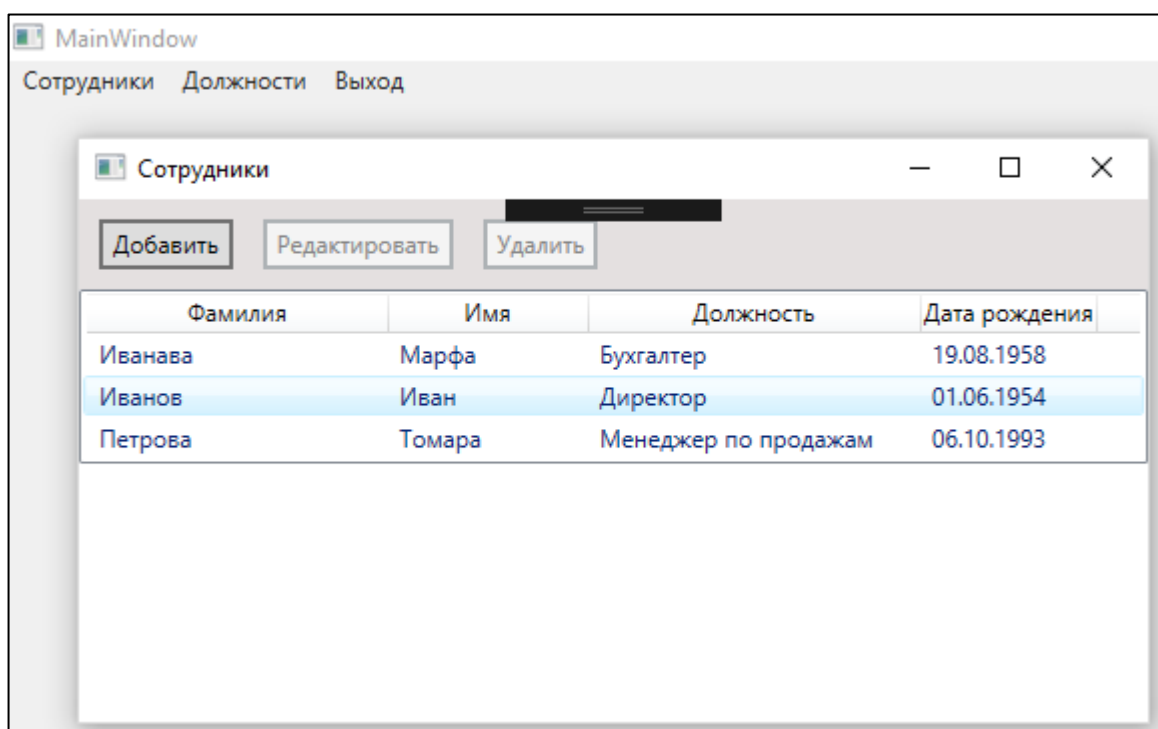


Рисунок 7.23 – Обновленные данные после удаления

Задание на лабораторную работу

1. Сделайте копию проекта, созданного в лабораторной работе 7.
2. Создайте EDM-модель данных для вашего приложения.
3. Модифицируйте ваше приложение в соответствии с методикой изложенной в описании данной лабораторной работы.
4. Протестировать изменения в функционировании приложения.

ЛАБОРАТОРНАЯ РАБОТА 8. Модульное тестирование приложения

Цель работы: Получить навыки разработки приложений, взаимодействующих с данными, представленными моделью EDM.

Общие сведения о модельном тестировании

Рассмотрим пример применения технологии разработки через тестирование при создании приложения работы с банковским счетом.

1. Создадим решение SolutionDemoBankAccount.
2. В решение добавим проект библиотеки классов C# - BankAccount. В созданном проекте удалим класс Class1.cs
3. В решение добавим проект модульного теста UnitTestBankAccount. Переименуем класс UnitTest.cs в класс UnitTestBankAccount.cs и добавим ссылку на проект BankAccount.

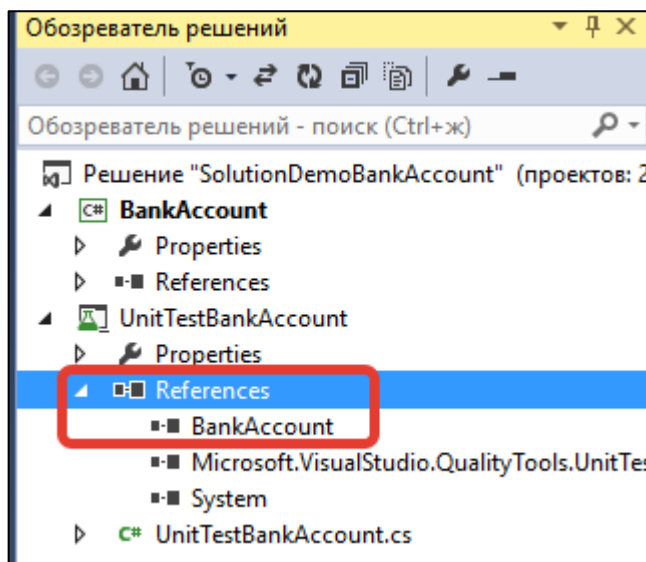


Рисунок 8.1 – Ссылка на проект BankAccount

4. В соответствии с требованиями в приложении должна быть возможность создания экземпляра класса банковского счета Account с актуализацией владельца и баланса счета. Для реализации этого требования создадим модульный тест конструктора класса Account с параметрами customer – полное имя клиента и balance – баланс счета.

```
[TestMethod]
public void TestCreateAccount()
{
    string customer = "Иванов Иван";
    double balance = 150.0;
    Account account = new Account(customer, balance);
    Assert.AreEqual(account.Customer, customer);
    Assert.AreEqual(account.Balance, balance);
}
```

Для созданного кода теста Visual Studio выделяет красным цветом несуществующие объекты.

- Для создания класса Account в проекте BankAccount выделите (рисунок 8.2) тип Account (1) в методе UnitTestBankAccount, в контекстном меню выберите пункт Сформировать (2) и в появившемся меню пункт Создать тип (3). В диалоговом окне Сформировать новый тип (рисунок 8.3) выберите проект BankAccount и нажмите кнопку ОК. В результате в проекте BankAccount будет сформирован шаблон для класса Account.

```
public class Account
{
}
```

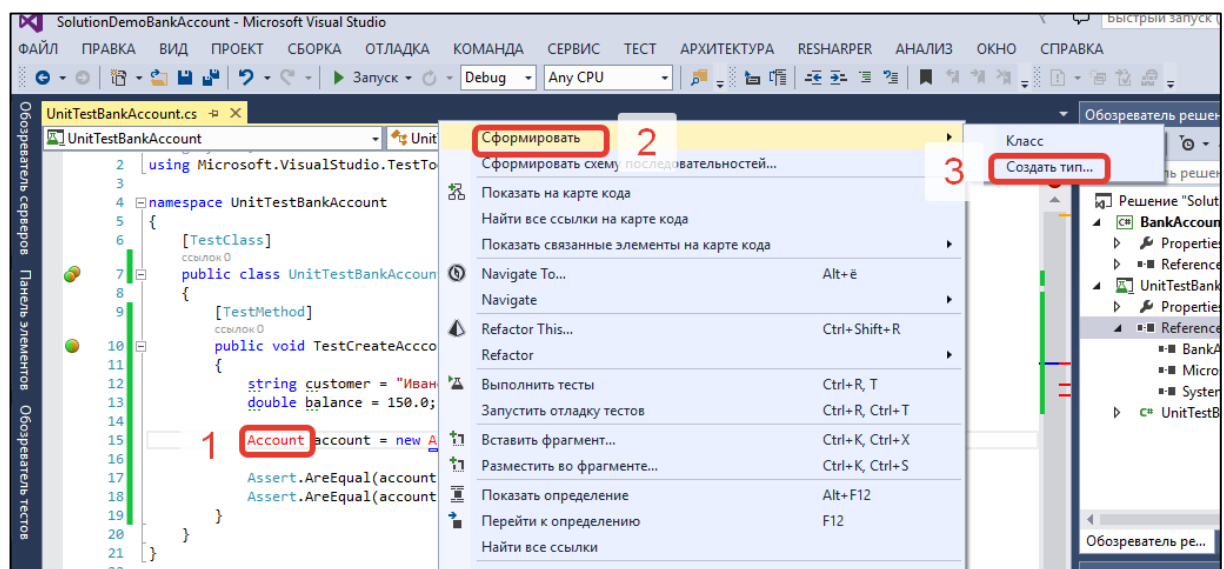


Рисунок 8.2 – Формирование класса Account

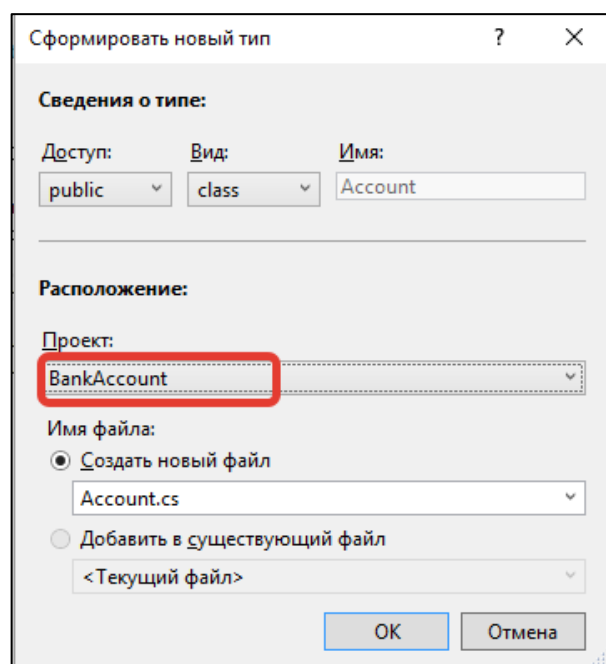


Рисунок 8.3 – Указание проекта BankAccount

6. На предыдущем шаге в проекте BankAccount был создан шаблон класса Account с пустым содержанием. Для генерации конструктора с параметрами выделите параметр конструктора класса Account (1) в методе UnitTestBankAccount, в контекстном меню выберите Сформировать (2) и в открывшемся меню Конструктор (3), что показано на рисунке 8.4.

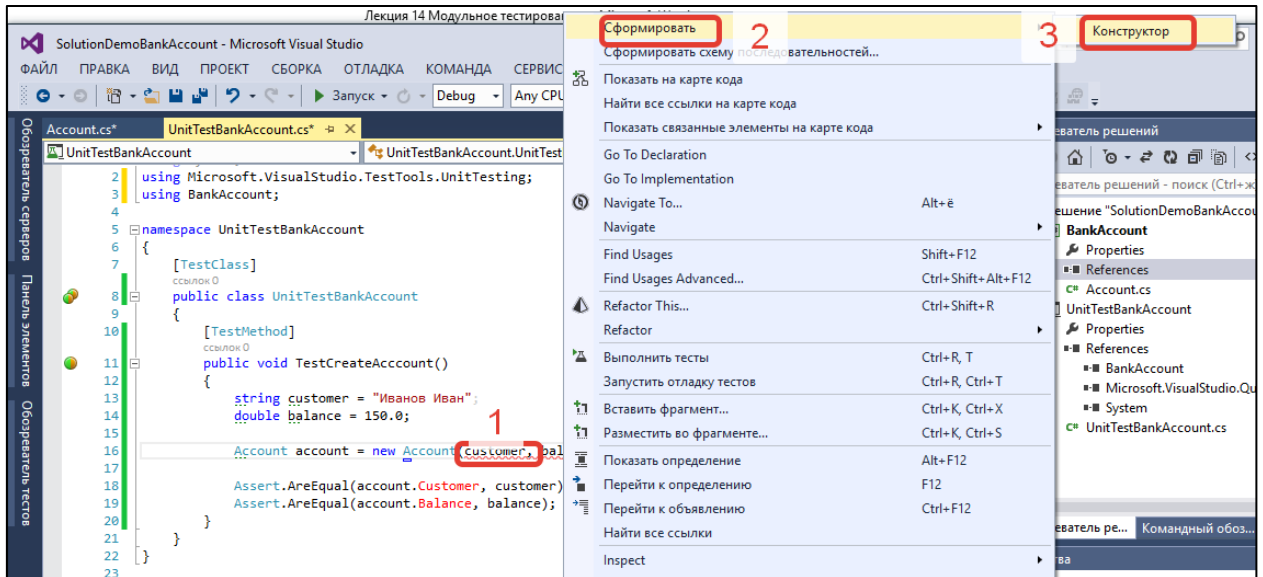


Рисунок 8.4 – Формирование конструктора класса

В результате для класса Account будут сгенерированы поля и конструктор с параметрами.

```
namespace BankAccount
{
    public class Account
    {
        private string customer;
        private double balance;
        public Account(string customer, double balance)
        {
            // TODO: Complete member initialization
            this.customer = customer;
            this.balance = balance;
        }
    }
}
```

7. В тестовом методе остаются неопределенными свойства класса Account. Для генерации свойств Customer и Balance класса Account сделайте инкапсуляцию полей customer и balance. Класс Account будет выглядеть следующим образом.

```
public class Account
{
    public string Customer { get; private set; }
    public double Balance { get; private set; }
    public Account(string customer, double balance)
```

```

{
    // TODO: Complete member initialization
    this.Customer = customer;
    this.Balance = balance;
}
}

```

8. После сборки и запуска тест конструктора с параметрами `TestCreateAccount` будет успешно выполнен (рис 8.5).

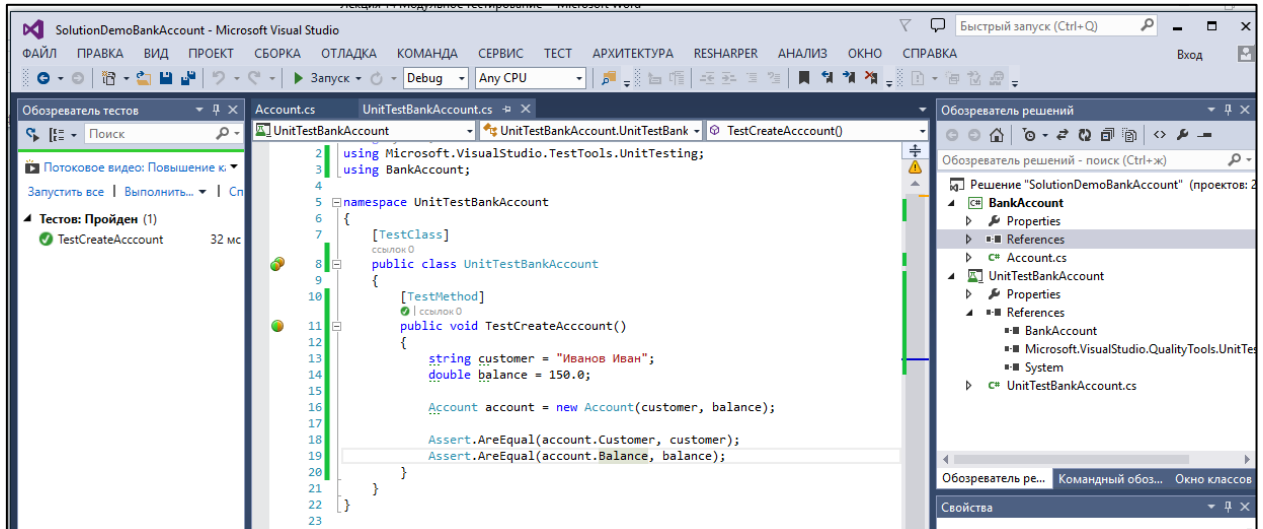


Рисунок 8.5 – Выполнение теста `TestCreateAccount`

9. Следующим шагом разработки приложения является добавление возможности зачисления денежных средств на счет клиента. Сформируем тестовый метод `TestCredit` для тестирования возможности зачисления денежных средств на счет клиента посредством метода `Credit`

```

[TestMethod]
public void TestCredit()
{
    string customer = "Иванов Иван";
    double balance = 150.0;
    Account account = new Account(customer, balance);
    double delta = 0.01;
    double amount = 12.52;
    double actualResult = 162.52;
    account.Credit(amount);
    double result = account.Balance;
    Assert.AreEqual(actualResult, result, delta);
}

```

В коде тестового метода `TestCredit` метод `account.Credit` класса `Account` не распознается, так как он на текущий момент отсутствует в классе `Account`. Сформируем заглушку метода `Credit` классе `Account`. Для этого выделим метод `account.Credit` в тестовом методе `TestCredit`, в кон-

текстом меню выберите Сформировать (2) и в открывшемся меню Заглушка метода (3), что показано на рисунке 8.6. В результате будет сформирован следующий код.

```
public void Credit(double amount)
{
    throw new NotImplementedException();
}
```

Теперь можно построить решение и выполнить тесты. Тест TestCredit не проходит (рисунок 8.7) и это является нормальной ситуацией, так как для методе Credit() определена только оболочка и отсутствует функциональность.

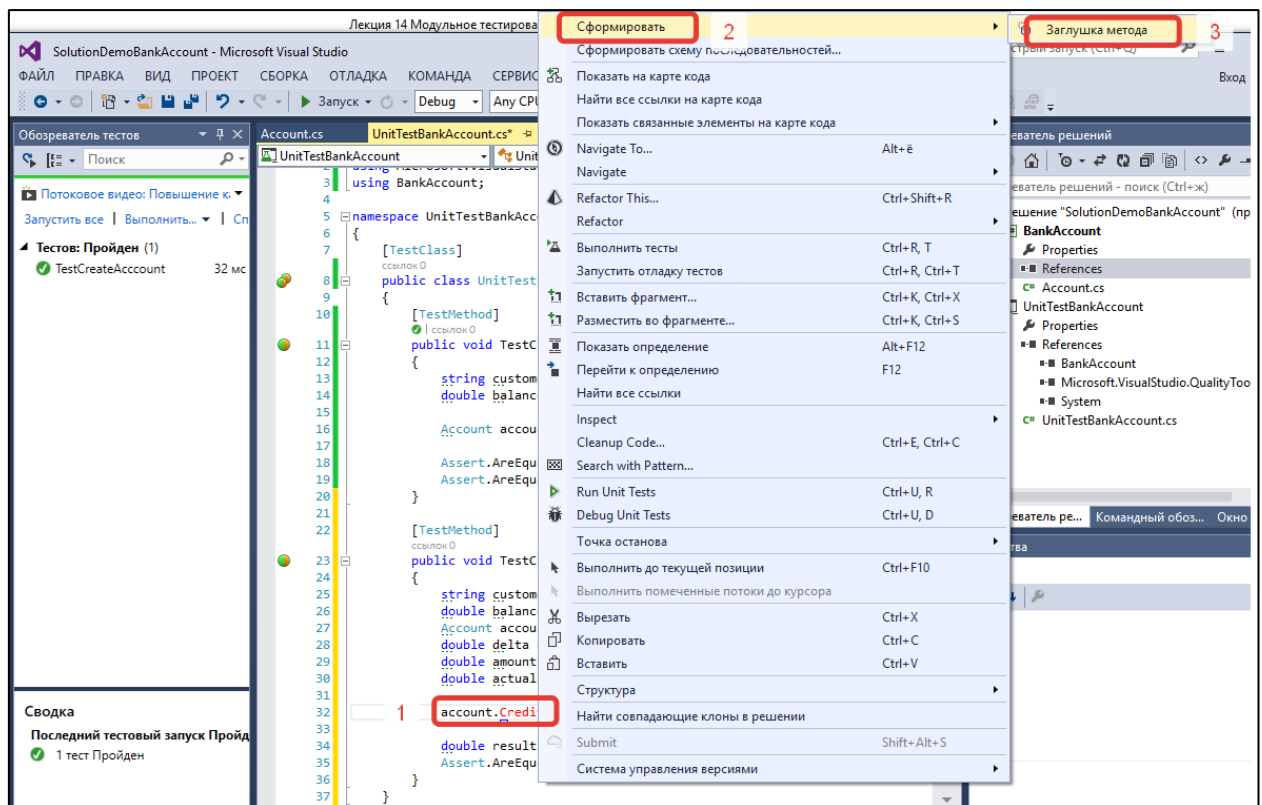


Рисунок 8.6 – Формирование заглушки метода Credit класса Account

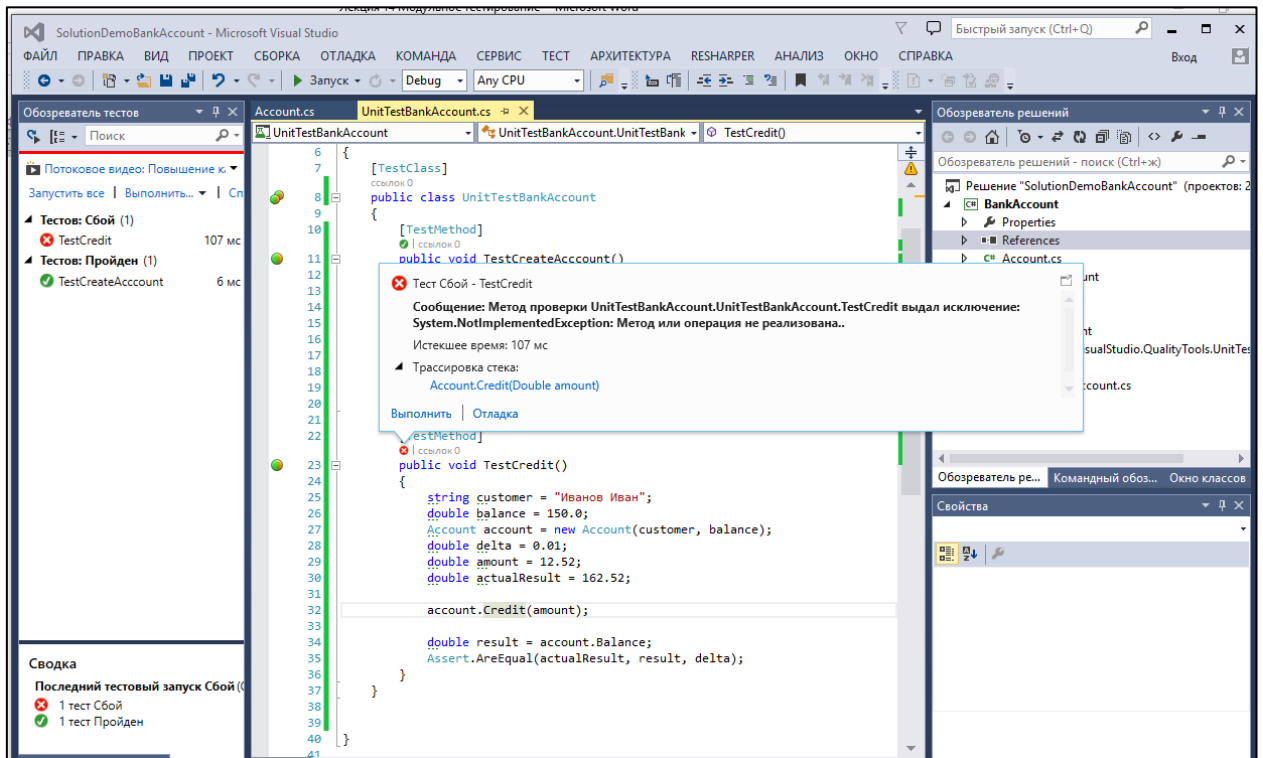


Рисунок 8.7 – Тест TestCredit не выполнен

10. Добавим возможность зачисления денежных средств на счет клиента с помощью метода `Credit()`.

```
public void Credit(double amount)
{
    Balance += amount;
}
```

После построения и запуска теста `TestCredit` получаем успешное завершение теста.

11. В соответствии с бизнес-правилами зачисляемая на счет клиента сумма не может быть меньше нуля. Для выполнения данного требования создадим тест `TestCreditWithAmountLessThanZero`.

```
[TestMethod]
[ExpectedException(typeof(ArgumentOutOfRangeException))]
public void TestCreditWithAmountLessThanZero()
{
    string customer = "Иванов Иван";
    double balance = 150.0;
    Account account = new Account(customer, balance);
    double delta = 0.01;
    double amount = -12.52;
    account.Credit(amount);
}
```

При создании теста предполагается, что если метод `Credit()` получает отрицательный параметр `amount` (-12.52) зачисления денежных средств, то в методе `Credit()` должно генерироваться исключение. Для теста `TestCreditWithAmountLessThanZero` применен атрибут

[ExpectedException(typeof(ArgumentOutOfRangeException))], который формирует успешное завершение теста, если при выполнении метода Credit() было сгенерировано исключение ArgumentOutOfRangeException. Если сейчас выполнить тест TestCreditWithAmountLessThanZero, то он завершится неудачей.

12. Внесем изменение в метод Credit(), который должны привести к генерации исключения при отрицательном значении параметра amount.

```
public void Credit(double amount)
{
    if (amount < 0)
    {
        throw new ArgumentOutOfRangeException("amount");
    }
    Balance += amount;
}
```

После сборки и запуска тест TestCreditWithAmountLessThanZero завершается успешно.

В рассмотренном примере с использованием *технологии разработки через тестирование* была реализована возможность зачисления денежных средств на счет клиента. По аналогии можно добавить возможность списания денежных средств клиента со счета, принимая во внимание заданные заказчиками бизнес-правила, а также другие возможности работы с банковским счетом клиента.

Задание на лабораторную работу

1. Сделайте копию проекта, созданного в лабораторной работе 7.
2. Создайте EDM-модель данных для вашего приложения.
3. Модифицируйте ваше приложение в соответствии с методикой изложенной в описании данной лабораторной работы.
4. Протестировать изменения в функционировании приложения.

ЗАКЛЮЧЕНИЕ

При написании данного лабораторного практикума ставилась цель предоставить студентам возможность закрепления теоретических знаний, изложенных в учебном пособии «Разработка и сопровождение программных систем. Учебное пособие», изданное в 2018 году авторами в издательстве РГЭУ. Лабораторный практикум способствует получению практических навыков по основным возможностям технологии Microsoft .NET для разработке программных приложений. В лабораторных работах рассмотрены основы технологии WPF, элементы управления пользовательского интерфейса, которые применяются для организации взаимодействия с пользователем. Свойства зависимостей и маршрутизируемые события, которые расширяют понятия свойств и событий языка C#, и повышают эффективность процессов связывания данных и обработки событий на разных логических уровнях приложения. Модель команд обеспечивает делегирование событий определенным командам и возможность повторного использования кода команд. Стили, ресурсы и шаблоны позволяют создать удобную систему управления поведением и визуализации интерфейсных элементов. Привязка данных обеспечивает связывание элементов управления и интерфейсных элементов с данными. Файловый ввод-вывод является важным элементом обеспечения взаимодействия приложения с внешней средой. Сериализация объектов представляет удобный инструмент сохранения и реконструкции программных объектов. Обработка исключения необходима при разработке и отладке программных приложений. Асинхронное программирование позволяет повысить эффективность работы программы при выполнении длительных операций. Модульное тестирование является важным компонентом разработчика, позволяющим повысить качество программных приложений. Шаблон MVVM решает задачи обеспечения качественного дизайна программных компонентов, распределения функциональности между логическими слоями, удобства тестирования, сопровождения и модификации.

Многие вопросы не вошли в данный лабораторный практикум. Это прежде всего использование средств разработки *Expression Design* и *Expression Blend*, работа со звуком и видео, анимация, деловая графика, вопросы интеграции приложений с другими системами. Технологии Microsoft постоянно развиваются, так на момент завершения работы над лабораторным практикумом вышел пакет обновлений *Visual Studio 2017* и ряда библиотек, расширяющих возможности создания программных систем.

По мнению авторов, данный лабораторный практикум может сформировать представление в возможностях технологий *WPF* для разработки приложений, а детальное изучение данных вопросов и получение практических навыков является прерогативой читателей.

Замечания, пожелания и ваше мнение по данному лабораторному практикуму можно направлять по адресу *alexdoljenko@mail.ru*.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Албахари Д., Албахари Б. С# 7.0. Справочник. Полное описание языка – М.: Издательский дом «Вильямс», 2013.
2. Васильев А.Н. Программирование на С# для начинающих. Основные сведения. – Изд.: Эксмо, 2018.
3. Вагнер Б. Наиболее эффективное программирование на С#. – Изд. «Альфа-книга», 2018.
4. Долженко А.И. Современные технологии программирования. Разработка приложений на базе WPF и SilverlightЖ учебник. – Рост. Гос. Эконом. Ун-т (РИНХ). – Ростов н/Д, 2011.
5. Долженко А.И. Разработка программных приложений на базе шаблона MVVM: учебное пособие. – Рост. Гос. Эконом. Ун-т (РИНХ). – Ростов н/Д, 2013.
6. Девис А. Асинхронное программирование в С# 5.0. – Изд. «ДМК», 2018.
7. Дейт К. Дж. Введение в системы баз данных. – М.: Издательский дом «Вильямс», 2002.
8. Джуст В. Разработка обслуживаемых программ на языке С#. – Изд. «ДМК ПРЕСС», 2018.
9. Долженко А.И. Технологии командной разработки программного обеспечения информационных систем. – М: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016.
10. Долженко А.И. Технологии программирования: учебник – Ростов н/Д: Редакционно-издательский комплекс Рост. гос. экон. Ун-та (РИНХ), 2016.
11. Долженко А.И., Глушенко С.А. Разработка и сопровождение программных систем. Технология Microsoft.NET для разработки приложений: учеб. пособие / А.И. Долженко, С.А. Глушенко. – Ростов р/Д: Издательско-полиграфический комплекс РГЭУ (РИНХ), 2018.
12. Натан А. WPF 4. Подробное руководство / А. Натан. – Спб.: Символ-Плюс, 2012. – 880 с.
13. Нейгел К. Professional C# 5.0 and .NET 4.5 / К. Нейгел, Б. Ивсен, Дж. Глинн, К. Уотсон.: Пер. с англ. – М.: Издательский дом «Вильямс», 2013.
14. Ошероув Р. Искусство автономного тестирования с примерами на С#. Второе издание. – Изд. «ДМК», 2014.
15. Петкович Д. Microsoft® SQL Server™ 2012. Руководство для начинающих: Пер. с англ. — СПб.: БХВ-Петербург, 2013
16. Петцольд Ч. Microsoft Windows Presentation Foundation. Базовый курс.: - М.: Изд. «Русская Редакция» СПб.: Питер, 2012.
17. Стиллмен Э Изучаем С#. – Изд. «ПИТЕР», 2012.
18. Свойства. [Электронный ресурс] <https://docs.microsoft.com/ru-ru/dotnet/framework/wpf/advanced/properties-wpf>

- 19.События. [Электронный ресурс] <https://docs.microsoft.com/ru-ru/dotnet/framework/wpf/advanced/events-wpf>
- 20.Троелсен, Э. Джепикс, Ф Язык программирования C # 7 и платформы .NET и .NET Core, 8-е изд.: Пер. с англ. –М.: Издательский дом «Вильямс», 2018
- 21.Универсальные приложения для Windows и Windows Phone [Электронный ресурс] <http://habrahabr.ru/company/microsoft/blog/218441/>
- 22.Data Binding . [Электронный ресурс] [https://docs.microsoft.com/en-us/previous-versions/windows/silverlight/dotnet-windows-silverlight/cc278072\(v=vs.95\)](https://docs.microsoft.com/en-us/previous-versions/windows/silverlight/dotnet-windows-silverlight/cc278072(v=vs.95))
- 23.WPF: Windows Presentation Foundation в .NET 4.5 с примерами на C# 5.0 для профессионалов.: Пер. с англ. –М.: Издательский дом“Диалектика/Вильямс”, 2018.
- 24.XAML в WPF. [Электронный ресурс] <https://docs.microsoft.com/ru-ru/dotnet/framework/wpf/advanced/xaml-in-wpf>

Учебное издание

Разработка и сопровождение программных систем.
Технологии Microsoft.NET для разработки приложений

Лабораторный практикум

Алексей Иванович Долженко, Сергей Андреевич Глушенко

Редактор Саркисова Е.В.
Корректор Петросян И.В.
Выпускающий редактор Акимова Л.И.
Директор издательства

Изд. № . Подписано к печати .
Объем уч.-изд. л. Гарнитура «Times New Roman»
Заказ . Тираж 100 экз.

344002, г. Ростов-на-Дону, ул. Б. Садовая, 69, РГЭУ (РИНХ).
Отпечатано в типографии РИЦ РГЭУ (РИНХ)