МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«МОСКОВСКИЙ АВТОМОБИЛЬНО-ДОРОЖНЫЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ (МАДИ)»
ВОЛЖСКИЙ ФИЛИАЛ

Кафедра гуманитарных и естественнонаучных дисциплин

Методические указания к лабораторным работам
по дисциплине

ВИЗУАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Направление подготовки

09.03.01 Информатика и вычислительная техника

Направленность (профиль, специализация) образовательной программы

«Автоматизированные системы обработки информации и управления»

Квалификация

бакалавр

Чебоксары
2019

Оглавление

Лабораторная работа №1

Создание главного окна приложения в среде C#

Цель работы: Изучить основные элементы среды разработки *Visual Studio Integrated Development Envronment* (*IDE* - интегрированная *среда разработки*) *C#* при создании на языке *C#* приложений с графически интерфейсом.

Основные сведения

Среда разработки Visual Studio Integrated Development Environment (IDE) - интегрированная *среда разработки*) включает набор инструментов и не зависит от используемых языков программирования, представленных в *Visual Studio*. *Visual Studio* можно использовать для создания кода и на различных языках программирования: управляемый *C++ - Manadged C++, Visual Basic.NET, Java.NET, C#.*

В лабораторной работе проводится изучении среды разработки на языке программирования *C#* и следующих средств проектирования *Windows* - приложений:

основные окна среды разработки *C#*;

- построение базовой инфраструктуры с помощью *Application Wizard* (мастер создания приложений);
- использование дизайнера форм *Dialog Painter* (программа для рисования диалоговых окон) для оформления диалоговых окон;
- добавление новых функциональных возможностей в приложение с использованием вкладки *Properties* (свойства).

Обзор среды разработки C#

Для начала работы с *Visual Studio.NET* необходимо из главного *меню* выбрать *пункт "Microsoft Visual Studio.NET" (VS)*. При этом на компьютере загрузится *Developer Studio* (визуальная *среда разработки Microsoft Visual*) на экране компьютера будет выведено окно, изображенное на рисунке 1.1.

Проектирование приложения

В качестве приложения разработаем простое *приложение, пользовательский интерфейс* которого будет содержать только главное окно. Для этого необходимо выполнить следующие шаги:

1. Создать рабочую область, называемую также рабочей средой (проектирования), рабочим пространством и рабочей обстановкой нового проекта.
2. Для создания каркаса приложения можно использовать мастер создания приложений - *Application Wizard*.
3. Изменить внешний вид автоматически создаваемых мастером окон до желаемого вида.

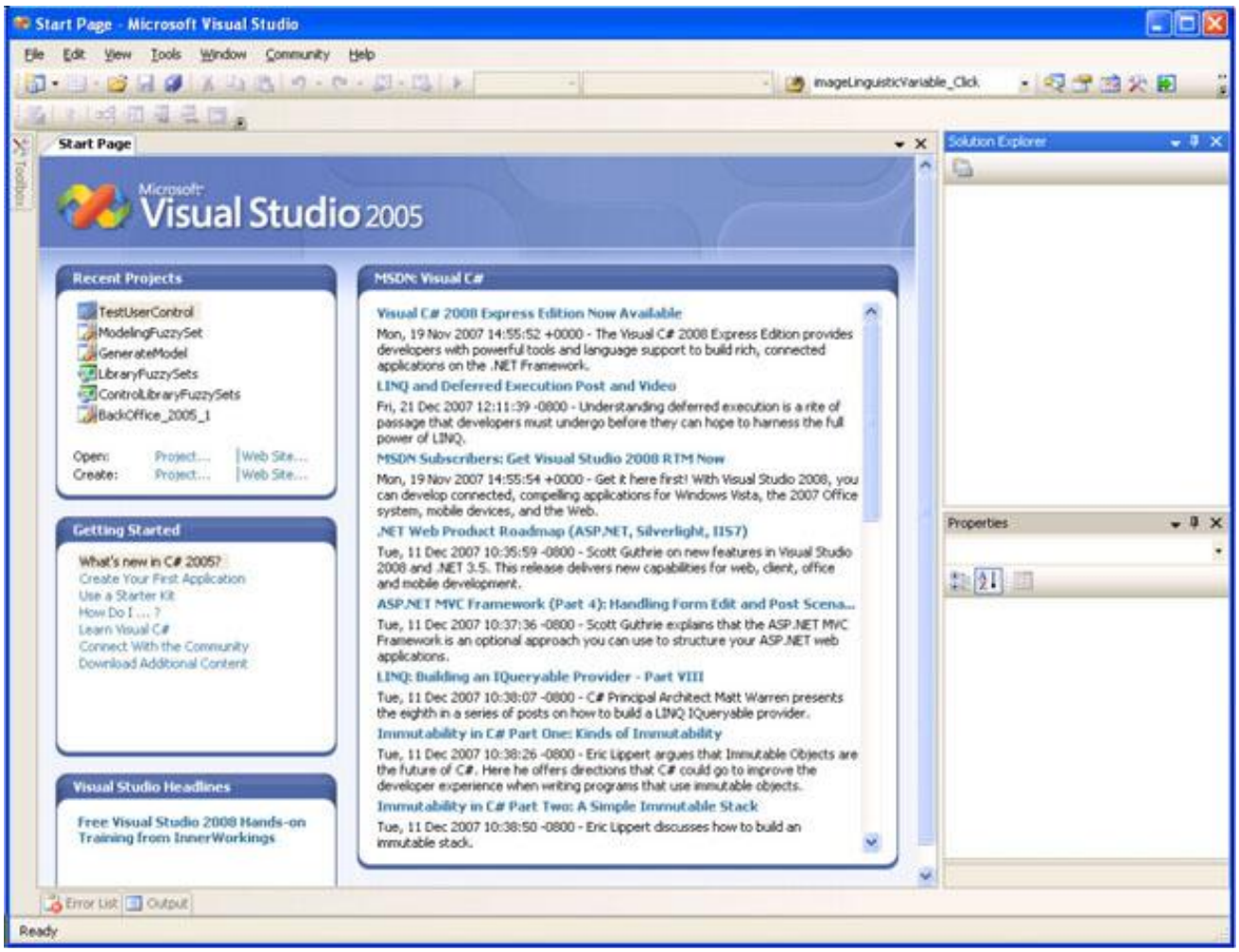

Рис. 1.1. Стартовое диалоговое окно IDE

4. Добавить код C#, который будет вызывать отображение приветствия.

Создание рабочей области проекта

В *VS* каждому разрабатываемому приложению нужна рабочая среда. Рабочая среда проекта состоит из папок, в которых хранятся файлы исходного кода, а также из папок, в которых хранятся различные конфигурационные файлы. Создание рабочей среды нового проекта производится следующим образом:

1. Щелкните на ссылке *Project* (Создать новый проект) метки *Create* на начальной странице *(Start Page) VS.NET*. При этом откроется окно создания нового проекта *New Project* (рисунок 1.2).

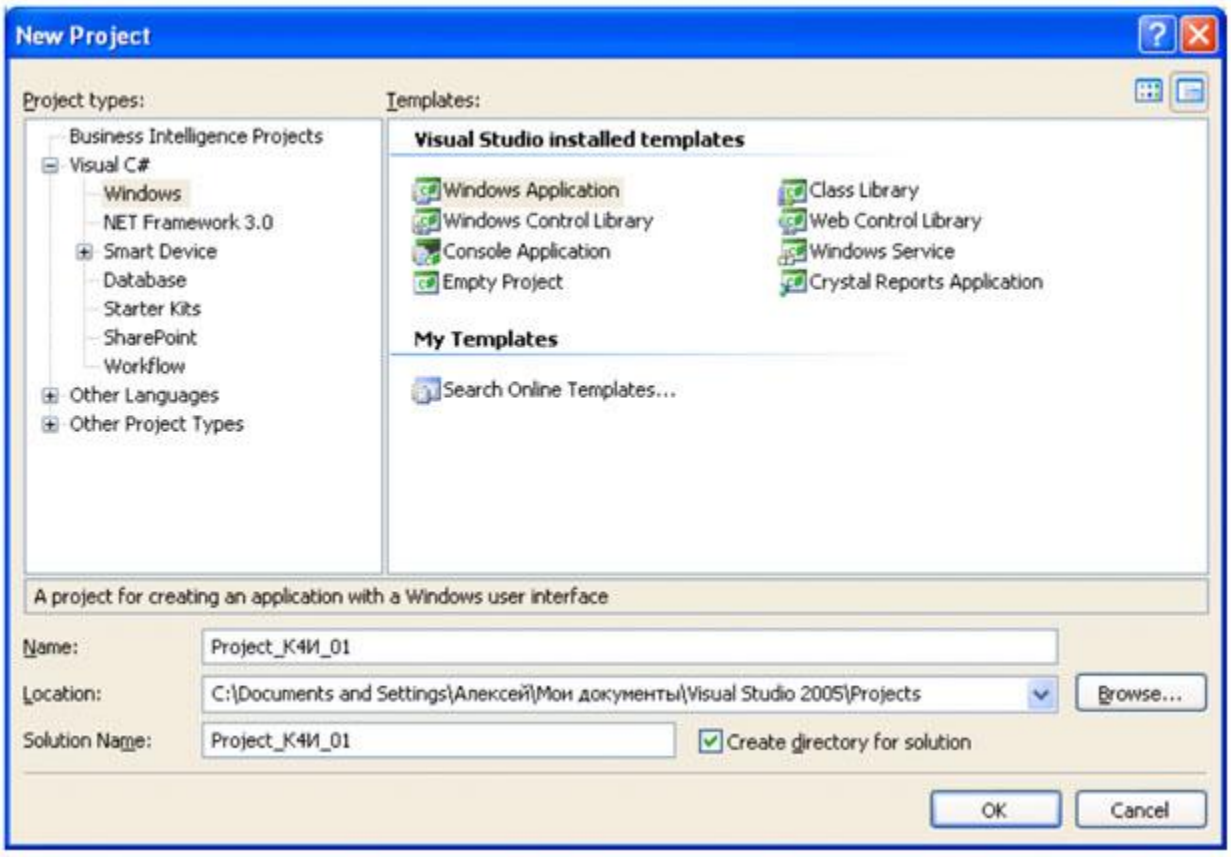

Рис. 1.2. Мастер создания нового проекта (New Project Wizard)

2. В дереве, отображаемом в подокне *Project Type* (Типы проектов) выберите *"Visual C# /Windows"*. В подокне *Templates* (Шаблоны) выберите *Windows Application* (Приложение *Windows*).
3. В поле *Name* (Название проекта) наберите имя проекта - Project_К4И_01 (имя проекта присваивается в соответствии со следующим синтаксисом: Project_"номер группы"_"номер бригады в группе").
4. Щелкните на кнопке *OK*. (Да). Мастер создания нового проекта создаст новый класс Form1, производный отSystem.Windows.Forms.Form с правильно настроенным методом Main( ). В свойствах проекта автоматически будут созданы ссылки на необходимые сборки библиотеки базовых классов. На экране появится графический шаблон среды разработки (рисунок 1.3).

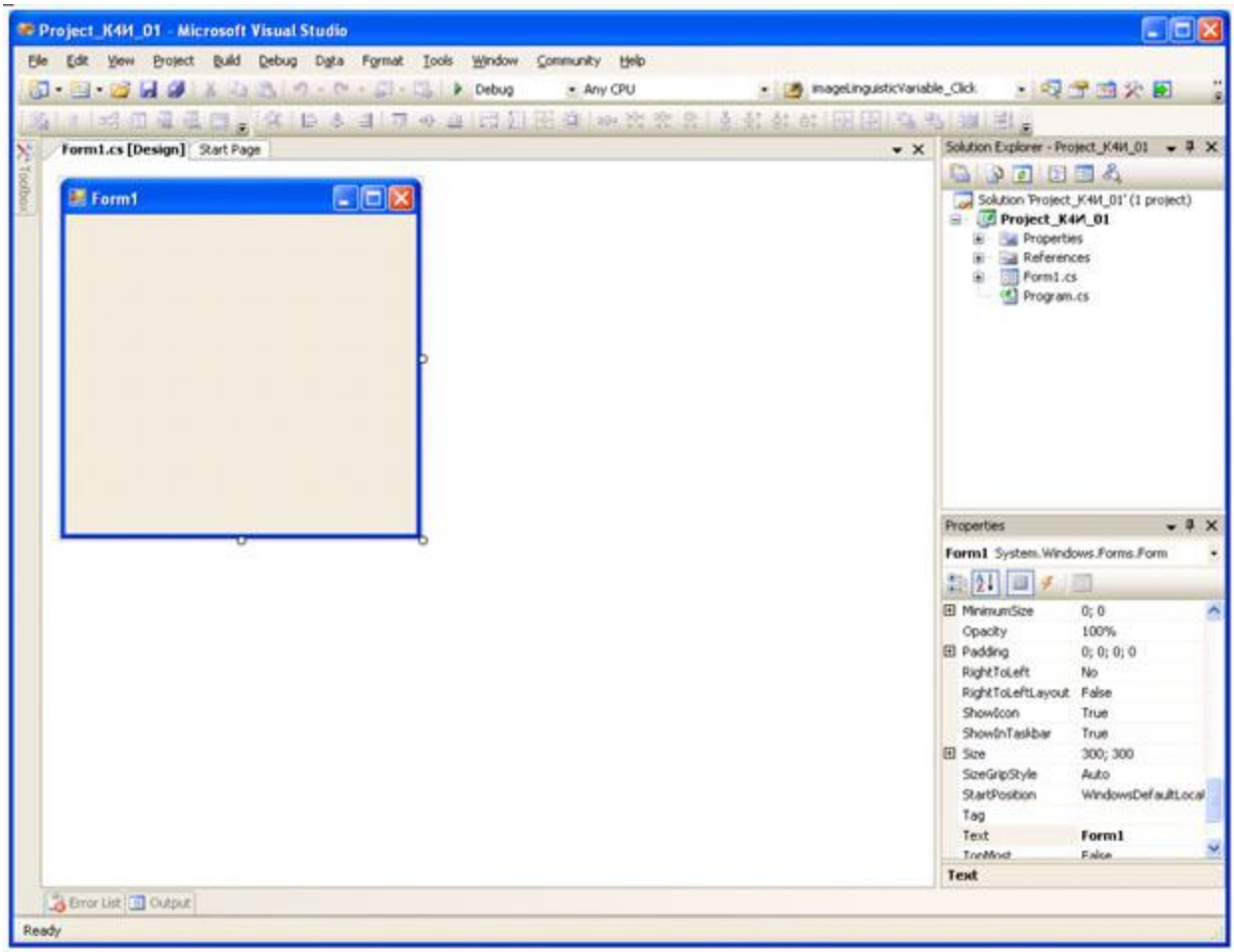

Рис. 1.3. Графический шаблон главного окна приложения

При помощи дизайнера графических форм можно добавлять в *приложение* любые *элементы управления* и он будет автоматически генерировать код для этих элементов (по умолчанию *файл* с главной формой приложения называется Form1.cs).

Для просмотра кода сгенерированного приложения можно в окне *Solution Explorer* щелкнуть правой кнопкой мыши на файлеForm1.cs и в контекстном *меню* выбрать *View Code* (рисунок 1.4).

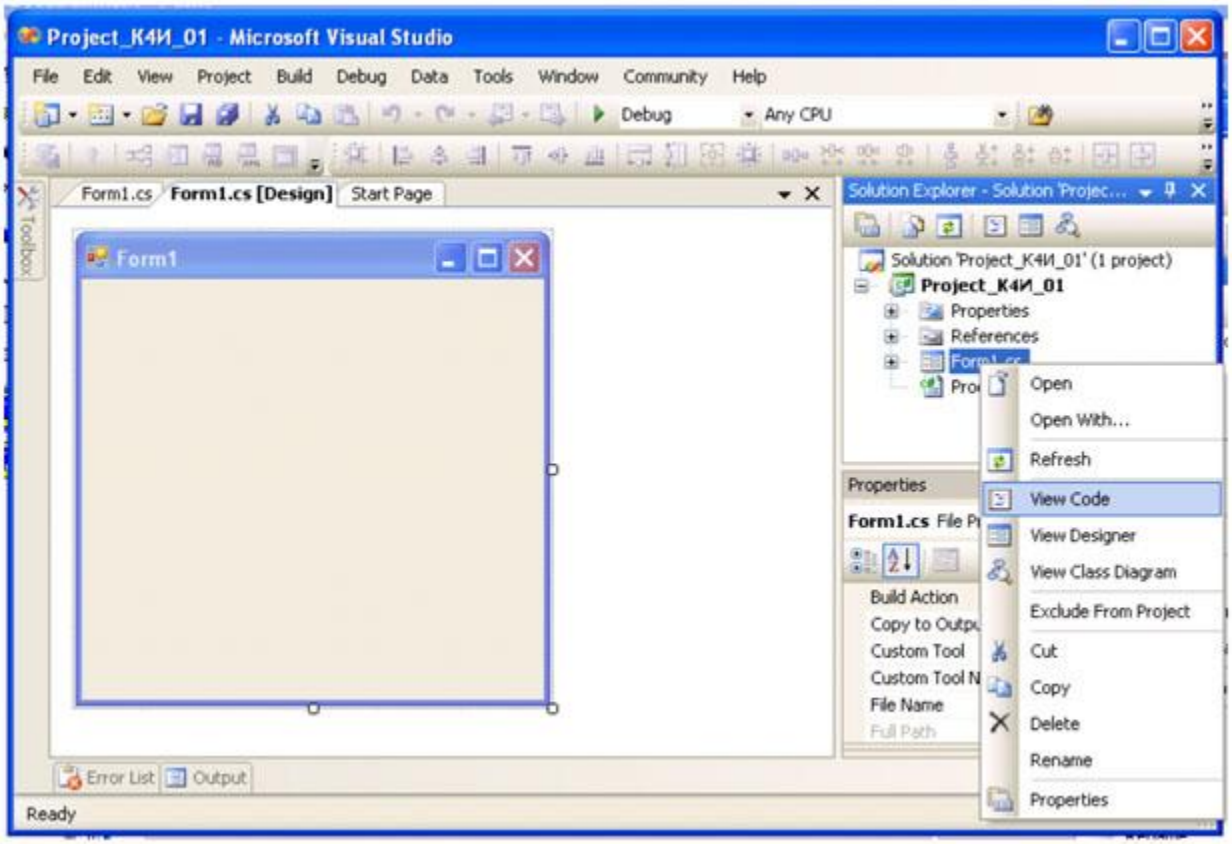

Рис. 1.4. Выбор режима View Code в контекстном меню

В результате будет выведен на экран следующий листинг кода приложения

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
namespace Project_К4И_01
{
 public partial class Form1 : Form
 {
  public Form1()
 { InitializeComponent(); }
 }
}
```

Инициализация компонент реализуется кодом, который можно отобразить, в окне *Solution Explorer* щелкнуть на пункте*Form.Designer.cs* (рисунок 1.5).

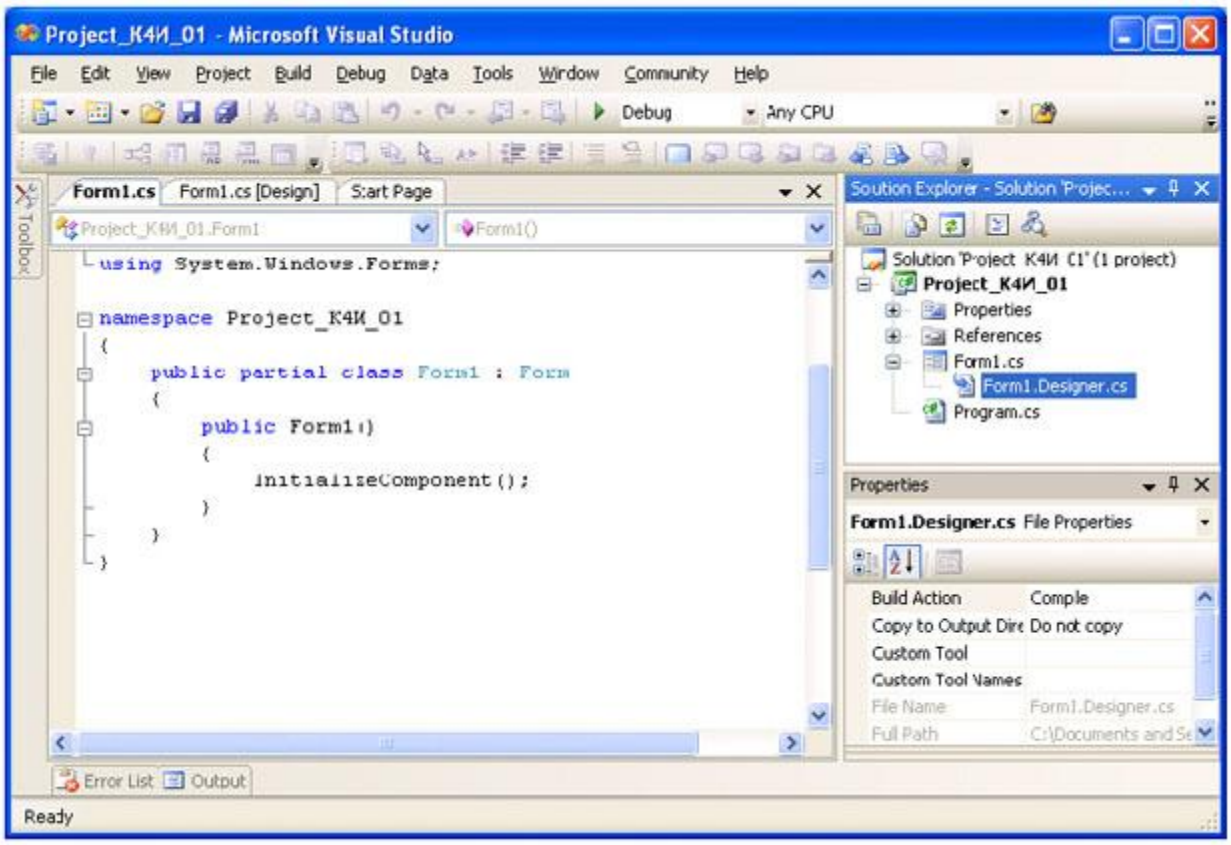

Рис. 1.5. Выбор режима Form.Designer.cs

```
namespace Project_К4И_01
{
 partial class Form1
 {
  private System.ComponentModel.IContainer components = null;
  protected override void Dispose(bool disposing)
  {
    if (disposing && (components != null))
    {
      components.Dispose();
    }
    base.Dispose(disposing);
  }
  #region Windows Form Designer generated code
  private void InitializeComponent()
  {
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
   }
   #endregion
 }
}
```

В определении класса Form1 используется *ключевое слово* partial, которое позволяет определять *класс*, структуру или*интерфейс*, распределенные по нескольким файлам. В *Visual Studio 2005* классы *Windows* -форм формируются в двух файлах:Form1.cs и Form1.Designer.cs.

В файле Form1.Designer.cs присутствует код, сгенерированный дизайнером *Windows* -формы, а файле Form1.cs - присутствует код инициализации класса и пользовательские члены класса (поля, свойства, методы, события,*делегаты*)

Код приложения (Program.cs) имеет следующий вид:

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
namespace Project_К4И_01
{
  static class Program
   {
    [STAThread]
    static void Main()
    {
     Application.EnableVisualStyles();
     Application.SetCompatibleTextRenderingDefault(false);
     Application.Run(new Form1());
    }
   }
}
```

Метод Main является точкой входа для приложения и вызывает Application.Run, который создает *класс* Form1.

Класс System.Windows.Forms.Application

Класс Application можно рассматривать как "*класс* низшего уровня", позволяющий нам управлять поведением приложения*Windows Forms*. Кроме того, этот *класс* определяет набор событий уровня всего приложения, например закрытие приложения или простой центрального процессора.

Наиболее важные методы этого класса (все они являются статическими) перечислены в таблице 1.1.

Таблица 1.1. Наиболее важные методы типа Application

Метод класса	**Назначение Application**
AddMessageFilter( )	Эти методы позволяют приложению перехватывать сообщения RemoveMessageFilter( ) и выполнять с этими сообщениями необходимые предварительные действия. Для того чтобы добавить фильтр сообщений, необходимо указать класс, реализующий интерфейс IMessageFilter
DoEvents( )	Обеспечивает способность приложения обрабатывать сообщения из очереди сообщений во время выполнения какой-либо длительной операции. Можно сказать, что DoEvents() - это "быстрый и грязный" заменитель нормальной многопоточности
Exit( )	Завершает работу приложения
ExitThred( )	Прекращает обработку сообщений для текущего потока и закрывает все окна, владельцем которых является этот поток
OLERequired( )	Инициализирует библиотеки OLE. Можете считать этот метод эквивалентом *.NET* для вызываемого вручную метода Olelnitialize( )
Run( )	Запускает стандартный цикл работы с сообщениями для текущего потока

Класс Application определяет множество статических свойств (таблице 1.2), большинство из которых доступны только для чтения.

Таблица 1.2. Наиболее важные свойства типа Application

Свойство	**Назначение**
CommonAppDataRegistry	Возвращает параметр системного реестра, который хранит общую для всех пользователей информацию о приложении
CompanyName	Возвращает имя компании
CurrentCulture	Позволяет задать или получить информацию о естественном языке, для работы с которым предназначен текущий поток
CurrentlnputLanguage	Позволяет задать или получить информацию о естественном языке для ввода информации, получаемой текущим потоком
ProductName	Для получения имени программного продукта, которое ассоциировано с данным приложением
ProductVersion	Позволяет получить номер версии программного продукта
StartupPath	Позволяет определить имя выполняемого файла для работающего приложения и путь к нему в операционной системе

Многие из этих свойств предназначены для получения общей информации о приложении, такой как название компании, номер версии и т.п.

Таким образом, при помощи многих свойств (например, CompanyName или ProductName) можно очень просто получить*метаданные* уровня сборки. В сборке можно использовать любое количество встроенных и пользовательских атрибутов. В результате можно получить *значение* атрибута [assembly:AssemblyCompany(" ")] при помощи свойстваApplication.CompanyName без необходимости прибегать к использованию типов, определенных в пространстве именSystem.Reflection.

Проектирование окна приложения

Для разметки окон приложения в соответствии с требованиями пользователя необходимо изменить свойства класса Forms1. Это можно сделать с помощью дизайнера окон (*Form Designer*), путем изменения свойств в окне Свойства (*Properties*) или в коде программы.

Размеры окна можно изменить непосредственно в *Form Designer* с помощью мыши захватывая и, растягивая/сжимая границы окна.

Для изменения других свойств окна необходимо окно свойств *Properties*.

На вкладке *Properties* измените *значение* в *поле Text* (Заголовок) на *Проект К4И*. При этом на форме изменится заголовок окна (рисунок 1.6).

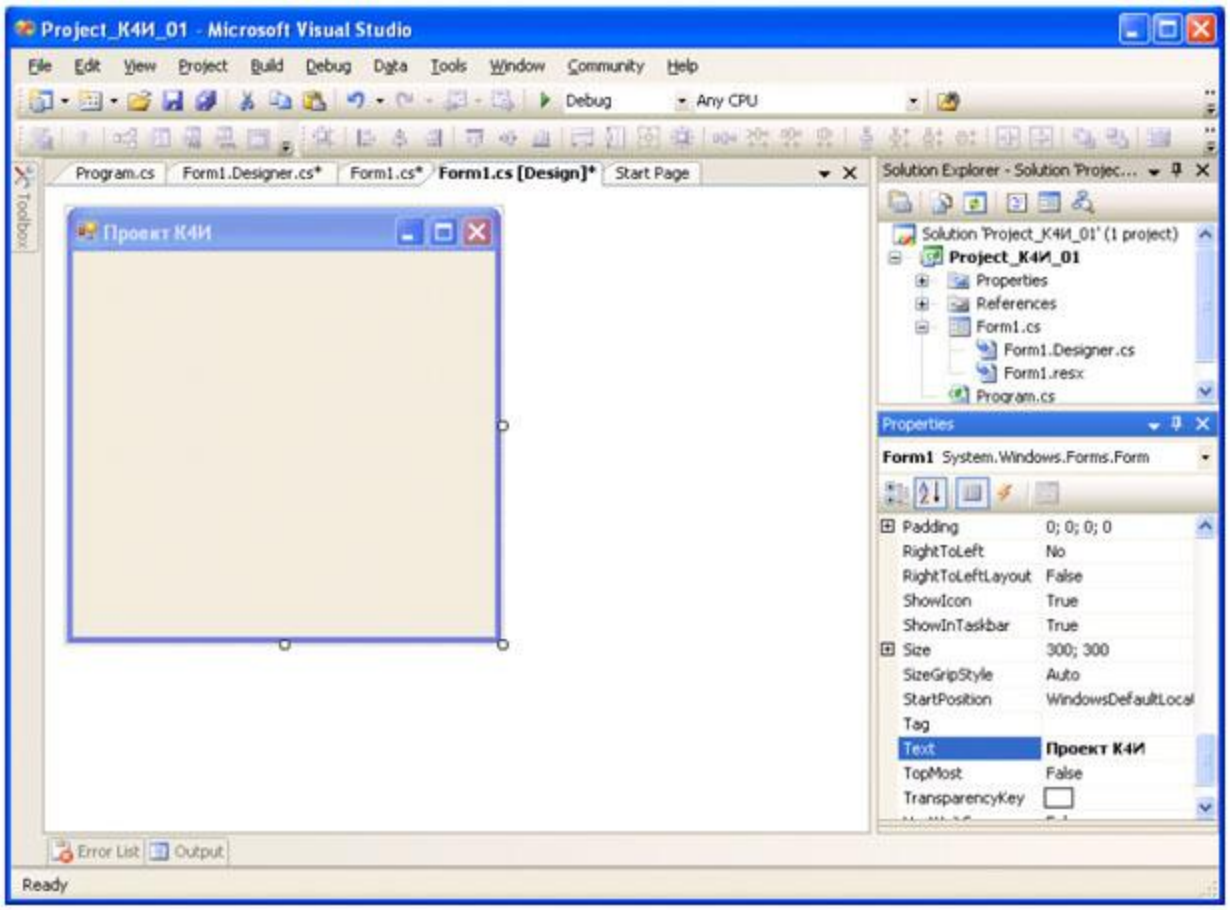

Рис. 1.6. Изменение значения в поле Text на вкладке Properties

Откомпилируйте *приложение*, выбрав из главного *меню* команду Build Project_К4И_01 (рисунок 1.7)

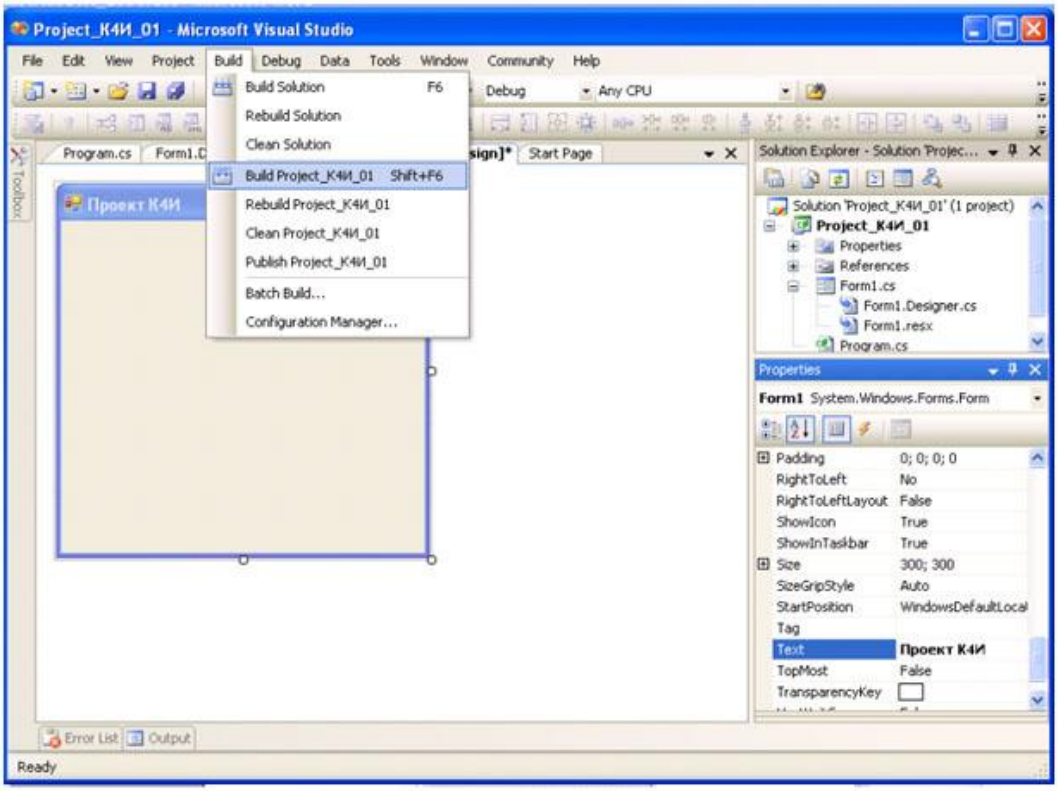

Рис. 1.7. Выбор из главного меню команды Build

В строке состояний должно появиться сообщение:
Build succeeded

Для запуска приложения выберите из главного *меню* команду Debug/Start (F5). *Приложение* запустится в отладочном режиме и на экране появится разработанное окно (рисунок 1.8.).

Рис. 1.8. Окно приложения Project_К4И_01

Для закрытия окна щелкните мышью на кнопке 

Добавление нового кода в приложение

Добавим в главную форму элемент контроля - кнопку. Для этого откроем вкладку *ToolBox* (рисунок 1.9) и сначала щелкнем мышью на элементе *Button* вкладки, а затем щелкнем мышью на форме.

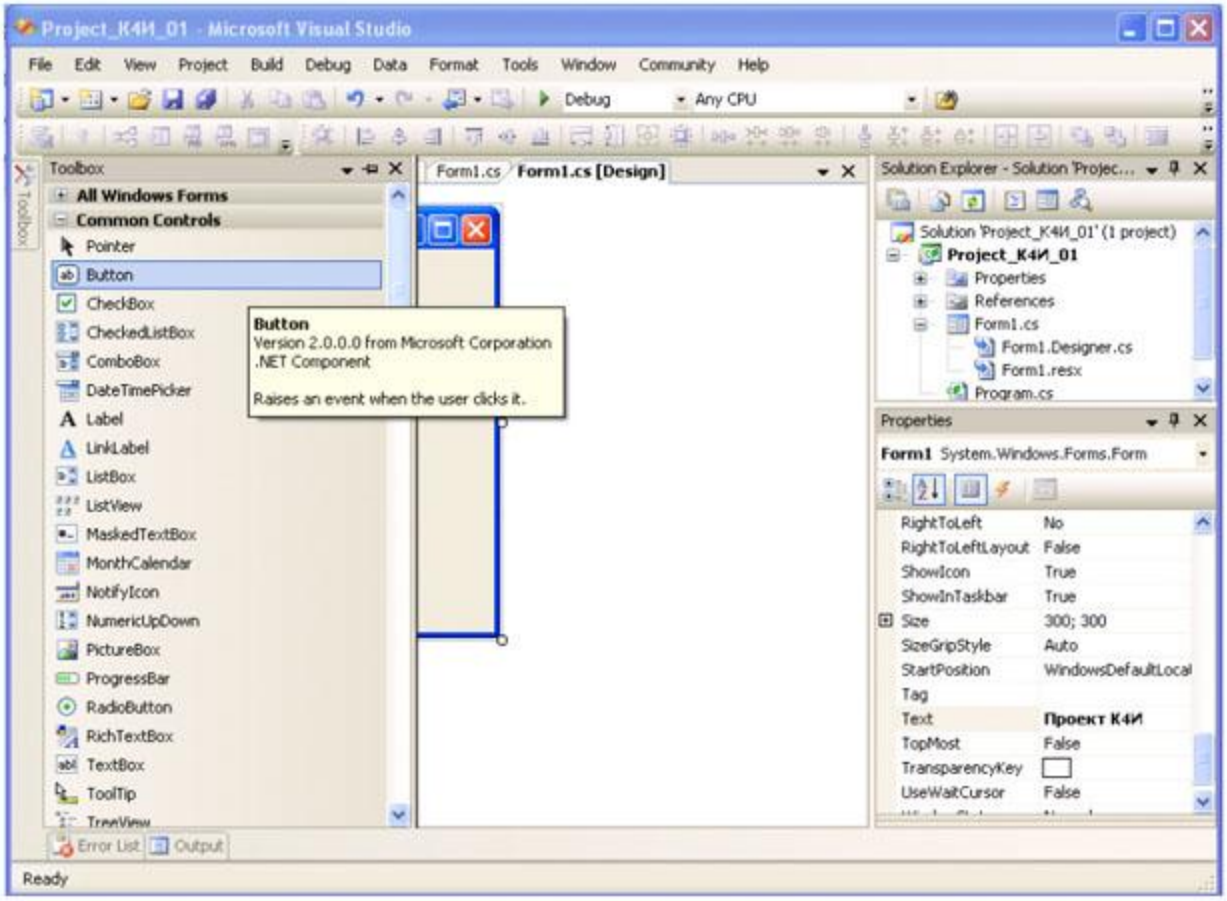

Рис. 1.9. Вкладка ToolBox

В результате получим форму с кнопкой (рисунок 1.10).

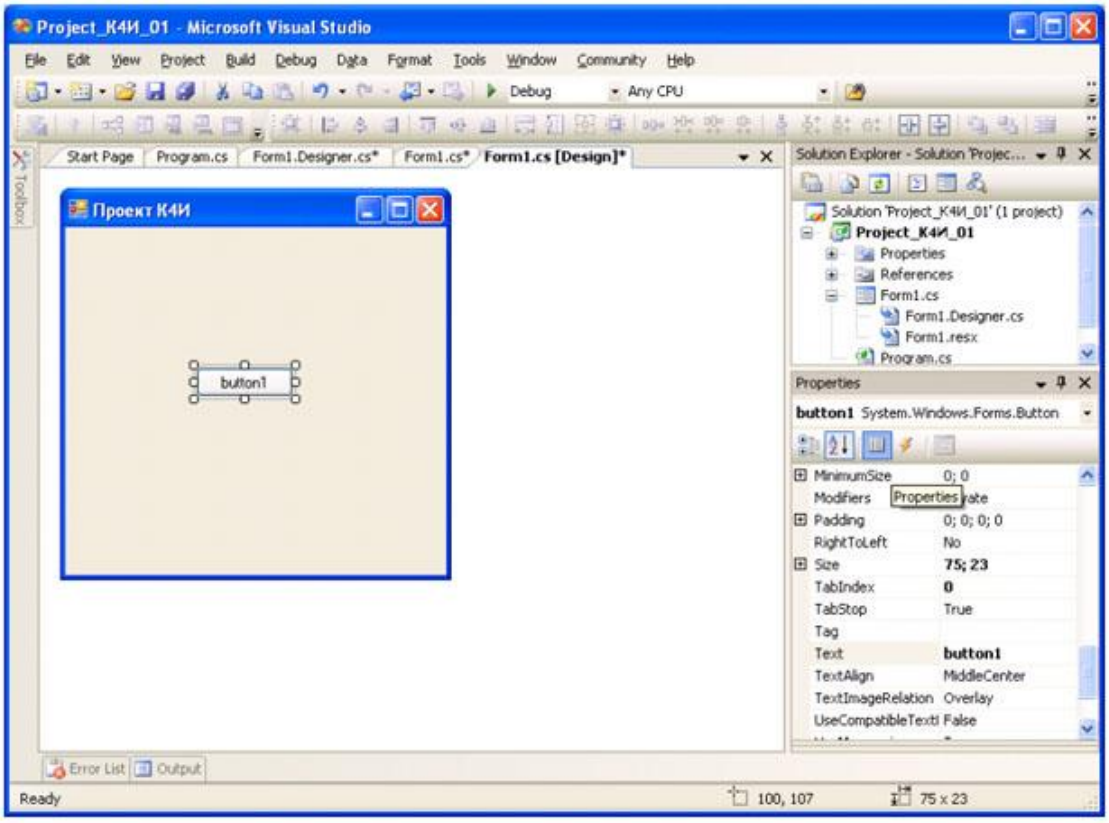

Рис. 1.10. Форма с установленной кнопкой

Установите кнопку в требуемое *место* на форме с помощью мыши.
Для задания текста на кнопке выделите ее на форме и откройте вкладку *Свойства* и измените свойство *Text* на "Приветствие". В результате название кнопки изменится (рисунок 1.11).

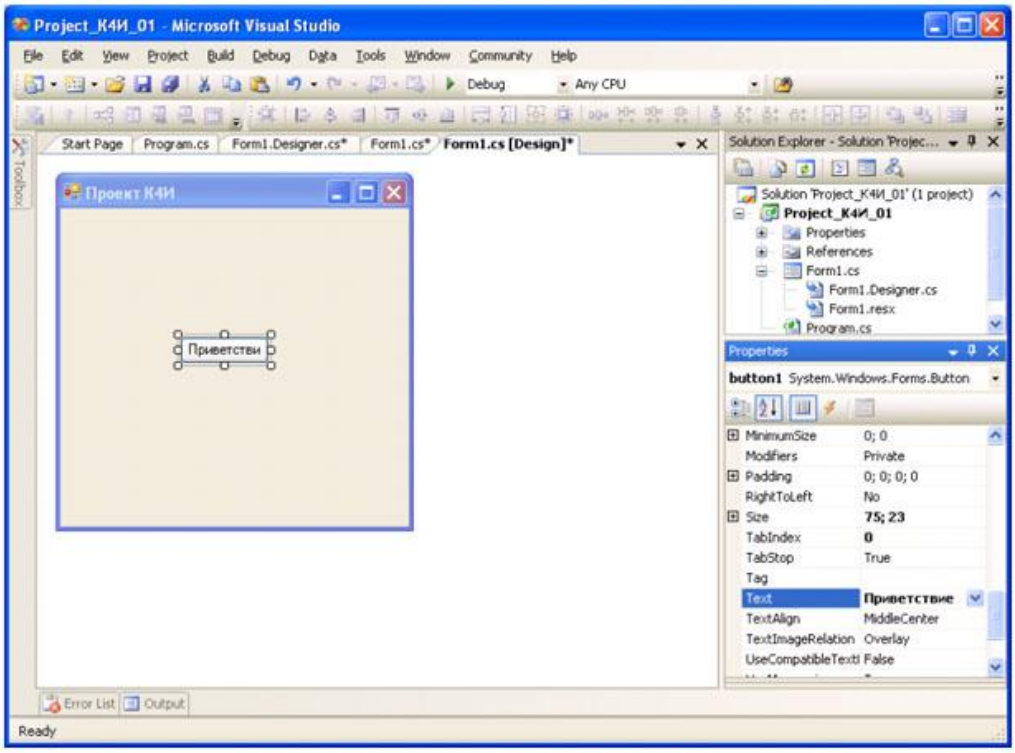

Рис. 1.11. Форма с измененным свойством Text кнопки

Для *связывания функций* кнопки с диалоговым окном необходимо создать обработчик события на нажатие кнопки. Для этого сделайте *двойной щелчок* на кнопке. В результате в коде приложения сформируется *шаблон* функции обработчика события Clickдля кнопки.

```
private void button1_Click(object sender, EventArgs e)
      {

      }
```

В полученный *шаблон* добавим функцию вывода диалогового окна с сообщением.

```
private void button1_Click(object sender, EventArgs e)
  {
   // Сообщение
   MessageBox.Show("Поздравляю с первым проектом на C#");
   }
```

После компиляции и запуска приложения получим следующее окно приложения (рисунок 1.12), а при нажатии кнопки будет выведено сообщение (рисунок 1.13).

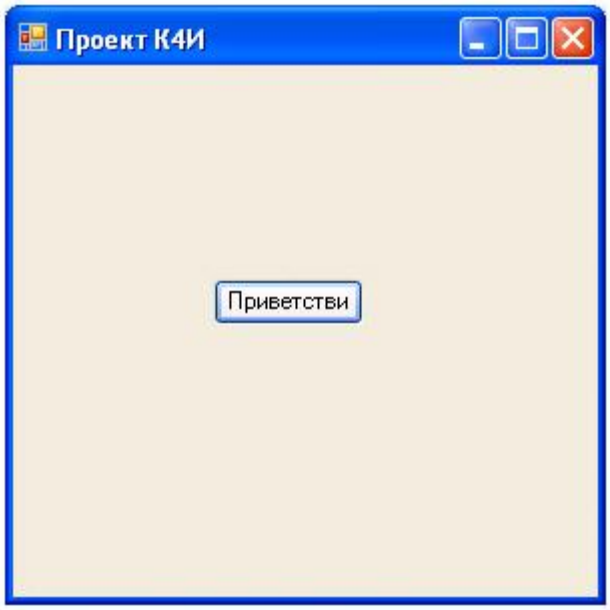

Рис. 1.12. Результат выполнения приложения

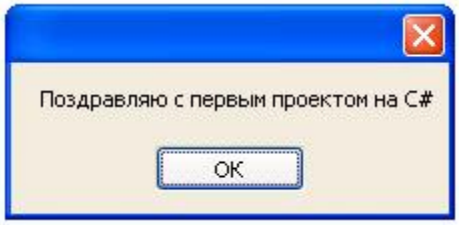

Рис. 1.13. Вывод сообщения

Структура и синтаксис функции

Рассмотрим код листинга функции:

```
private void button1_Click(object sender, EventArgs e)
  {
  // Сообщение
  MessageBox.Show("Поздравляю с первым проектом на C#");
  }
```

Первая строка является частью оболочки функции, сгенерированной *Developer Studio* на языке *C#*.

Первое *слово* в строке, private определяет видимость функции как внутреннюю, т.е. видимую только для членов класса Form1. Второе *слово* void, определяет *тип данных* возвращаемого значения (результата). Ключевое *слово* void, - перед именем функции, или в качестве аргумента функции (в скобках в конце строки), означает отсутствие соответствующего элемента. Третье*слово* в строке, button1_Click, обозначает *имя функции*. За именем функции следует *список* передаваемых ей аргументов, заключенный в круглые скобки. Круглые скобки нужно использовать всегда, даже когда у функции нет параметров.

Правило 1. В *C#* при вызове функции за ее именем должны стоять круглые скобки, даже если данной функции не передается ни один *параметр*.

В следующей строке листинга открывающая фигурная скобка ({) отмечает начало тела функции. В конце тела функции ставится закрывающая фигурная скобка (}).

Правило 2. *Тело функции* всегда заключается в *фигурные скобки* {}.

Следующая строка начинается с двух косых черт, или слешей (//). Все, что следует до конца строки после двух идущих подряд косых черт (без пробелов, табуляций и т.п. между ними) рассматривается компилятором как комментарий и игнорируется. Исключением являются строки, в которых косая черта является частью текстовой, или литерной строки (строки букв). Это один из способов комментирования кода. Второй способ чаще используется при добавлении в код нескольких строк комментариев. В этом случае начало комментария обозначается идущими подряд косой чертой и звездочкой (/*), а конец комментария завершается таким же набором символов, но переставленных в обратном порядке (*/).

Последняя строка в добавленном нами коде (в тексте это две строки):
MessageBox.Show("Поздравляю с первым проектом на C#");

Во-первых, *C#* чувствителен к регистру. В именах функций и переменных заглавные (прописные) буквы должны использоваться точно так же, как в их объявлениях. Это означает, что *компилятор* распознает следующие имена функций как имена трех различных функций:
MessageBox.Show messageBox.Show messagebox.Show

Правило 3. Язык *C#* чувствителен к регистру. При вводе программ, написанных на языке *C#*, учитывайте *регистр*. В частности, все идентификаторы вводите с учетом регистра.

За именем функции следуют аргументы функции, заключенные в круглые скобки, а после скобок стоит точка с запятой. Аргументы разделяются запятыми.

Задание на лабораторную работу

1. Изучить теоретический материал.
2. Создать Windows форму.
3. На Windows форме создать кнопку "Приветствие".
4. Протестировать работу приложения
5. Добавить в форму две кнопки (1 и 2), для которых задать различные цвета (свойство BackColor).
6. Написать для кнопок 1 и 2 обработчики, которые изменяют цвета кнопок: при неоднократном нажатии любой кнопки цвета кнопок меняются (цвет кнопки 1 меняется на цвет кнопки 2 и наоборот).
7. Добавьте кнопку "Выход". Закрытие приложения обеспечивает метод Exit( ) класса Application.
8. Протестировать работу приложения.

Лаораторная раота работа №2
Создание главного меню приложения

Цель работы: Изучить основные способы разработки главного *меню* приложении. Получить практические навыки в создании главного *меню* приложения.

Указания по использованию .NET

В любом языке программирования существуют традиционные стили программирования. Эти стили являются не частью самого языка, а соглашениями, скажем, по *именованию переменных* или использованию определенных классов, методов или функций. Если большинство разработчиков будут следовать одинаковым соглашениям, то им будет проще понять код друг друга, что, в свою*очередь*, облегчает поддержку программы. Так, общим соглашением в *Visual Basic 6* было то, что строковые переменные должны иметь имена, начинающиеся с s или str, например String sResult или String strMessage. Однако соглашения зависят от языка и среды разработки. Программисты на C++ для платформы *Windows* традиционно используют *префикс* psz или lpsz для обозначения строк: char *pszResult; char *lpszMessage;. Но на Unix-машинах такие префиксы не применяются: char *Result; char *Message;.

В соответствии с соглашениями в C# имена переменных не должны иметь префиксов: string Result; string Message;.

Соглашение, согласно которому имена переменных содержат *префикс*, указывающий *тип данных*, известно как "венгерский" стиль именования объектов. При чтении такого кода разработчики могут сразу же сказать по имени переменной, какой *тип данных* она представляет.

В то время как для многих языков соглашения по именованию вырабатывались одновременно с развитием языка, для C# и платформы *.NET Microsoft* написала подробные рекомендации по использованию, которые приведены в документации *MSDN* для*.NET/C#*. Следовательно, с самого начала программы *.NET* будут иметь более высокий уровень совместимости по части понимания кода другими разработчиками. Эти рекомендации были разработаны с учетом опыта, полученного на протяжении более двадцати лет объектно-ориентированного программирования, и в результате являются тщательно продуманными и хорошо восприняты сообществом разработчиков.

Однако необходимо отметить, что рекомендации не то же самое, что спецификации языка. Рекомендаций следует придерживаться по мере возможности. Если имеется веская причина для их несоблюдения, это не будет проблемой. Отклонение от рекомендаций должно быть вызвано реальными причинами, а не простым нежеланием.

Одним из важных моментов является выбор имен для элементов программы: переменных, методов, классов, перечислений и пространств имен.

Очевидно, что названия обязаны отражать назначение элемента и не должны конфликтовать с другими именами.

Общая философия платформы *.NET* состоит в том, что *имя переменной* должно отражать назначение экземпляра переменной, а не*тип данных*.

Например, Height - хорошее название, а IntegerValue - нет. Однако этот принцип является труднодостижимым идеалом. В частности, при работе с элементами управления в большинстве случаев вам будет удобнее использовать имена переменных, подобные ConfirmationDialog и ChooseEmployeeListBox.

Конкретные рекомендации по именованию включают в себя следующие *разделы*.

Практически во всех случаях для имен следует использовать стиль *Pascal*, при котором первая буква каждого слова в названии является прописной

Например: EmployeeSalary, ConfirmationDialog, PlainTextEncoding.

Соединение слов с помощью знака подчеркивания не приветствуется, поэтому не придумывайте такие имена, какemployee_salary. В других языках часто используют все прописные буквы в названиях констант. Это не рекомендуется в C#, поскольку такие имена трудно читать, лучше применять паскалевский стиль:
```
const int MaximumLength;
```

Еще одна рекомендуемая схема - именование в стиле *camel*. Именование *camel* аналогично паскалевскому стилю, за исключением того, что первая буква первого слова не является прописной: employeeSalary, confirmationDialog, plainTextEncoding.

Существуют две ситуации, в которых лучше применять такое именование. Имена всех параметров, передаваемых в методы, должны записываться в стиле *camel*:
```
public void RecordSale (string salesmanName,int guanuity);
```

Также можно использовать *camel* -соглашение для того, чтобы отличить два элемента, которые в противном случае имели бы одинаковые имена. Наиболее общий случай, когда свойство является оболочкой для поля.
```
private string employeeName;
public string EmployeeName
{ get
    { return employeeName; }
}
```

Приведенный код является совершенно корректным с точки зрения рекомендаций. Отметим, однако, что в этом случае следует применять соглашение *camel* для закрытых членов и соглашение *Pascal* для открытых или защищенных членов, чтобы другие классы, использующие ваш код, видели только имена в стиле *Pascal* (за исключением имен параметров).

В большинстве случаев следует применять соглашения *Pascal*. Тем не менее, соглашение *camel* рекомендуется для закрытых переменных, которые не видны вне класса, где две переменные имеют одинаковое назначение. Например, если есть publicсвойство, которое инкапсулирует private *поле* с тем же именем, то можно использовать соглашение *camel* для поля и соглашение*Pascal* для свойства, как в приведенном выше примере EmployeeName.

Также необходимо обращать внимание на чувствительность к регистру. C# чувствителен к регистру, поэтому синтаксически в C# допустимо, чтобы имена различались только регистром. Однако нужно помнить, что ваши сборки могут быть вызваны из приложений *VB.NET*, а *VB.NET* не является чувствительным к регистру. Поэтому

использовать имена, отличающиеся только регистром, можно лишь в том случае, если они никогда не будут видны вне сборки. В противном случае код, написанный в *VB.NET*, не сможет корректно использовать вашу сборку.

Необходимо по возможности делать так, чтобы стиль всех имен совпадал. Например, если один из методов в классе называетсяShowConfirmationDialog, то другому методу не следует давать имя ShowDialogWarning или WarningDialogShow. Он должен называться ShowWarningDialog.

Имена пространств имен следует выбирать особенно тщательно для того, чтобы избежать использования такого же имени, которое применяется где-то еще. Необходимо помнить, что *.NET* различает имена объектов в разделяемых сборках только по именам пространств имен. Если использовать для двух пакетов программного обеспечения одно и то же имя пространства имен и установить оба пакета на один *компьютер*, возникнут проблемы. Рекомендуется создавать *пространство имен* верхнего уровня с именем вашей компании, а затем вкладывать пространства имен, постепенно сужая их названия до технологии, группы или отдела, где вы работаете, или до названия пакета, для которого предназначены ваши классы. Microsoft рекомендует имена пространств имен, которые начинаются с <НазваниеКомпании>.<НазваниеТехнологии>, например,
WeaponsOfDestructionCorp.RayGunControllers

или
WeaponsOfDestructionCorp.Viruses.

Имена не должны конфликтовать с ключевыми словами. Если попытаться в программе назвать элемент по имени одного из ключевых слов C#, это практически всегда вызовет синтаксическую ошибку., так как *компилятор* предположит, что имя соответствует оператору.

Создание меню

В пространстве имен *System.Windows.Forms* предусмотрено большое количество типов для организации ниспадающих главных *меню*(расположенных в верхней части формы) и контекстных *меню*, открывающихся по щелчку правой кнопки мыши.

Элемент управления ToolStrip представляет собой *контейнер*, используемый для создания структур *меню*, панелей инструментов и строк состояний.

Элемент управления MenuStrip - это *контейнер* для структур *меню* в приложении. Этот элемент управлении наследуется отToolStrip. Система *меню* строится добавлением объектов ToolStripMenu к menuStrip.

Класс ToolStripMenuItem служит для построения структур *меню*. Каждый *объект* ToolStripMenuItem представляет отдельный*пункт* в системе *меню*.

Начнем с создания стандартного ниспадающего *меню*, которое позволит пользователю выйти из приложения, выбрав *пункт Объект > Выход*. Для этого необходимо перетащить элемент управления MenuStrip (рисунок 2.1) на форму в конструкторе.

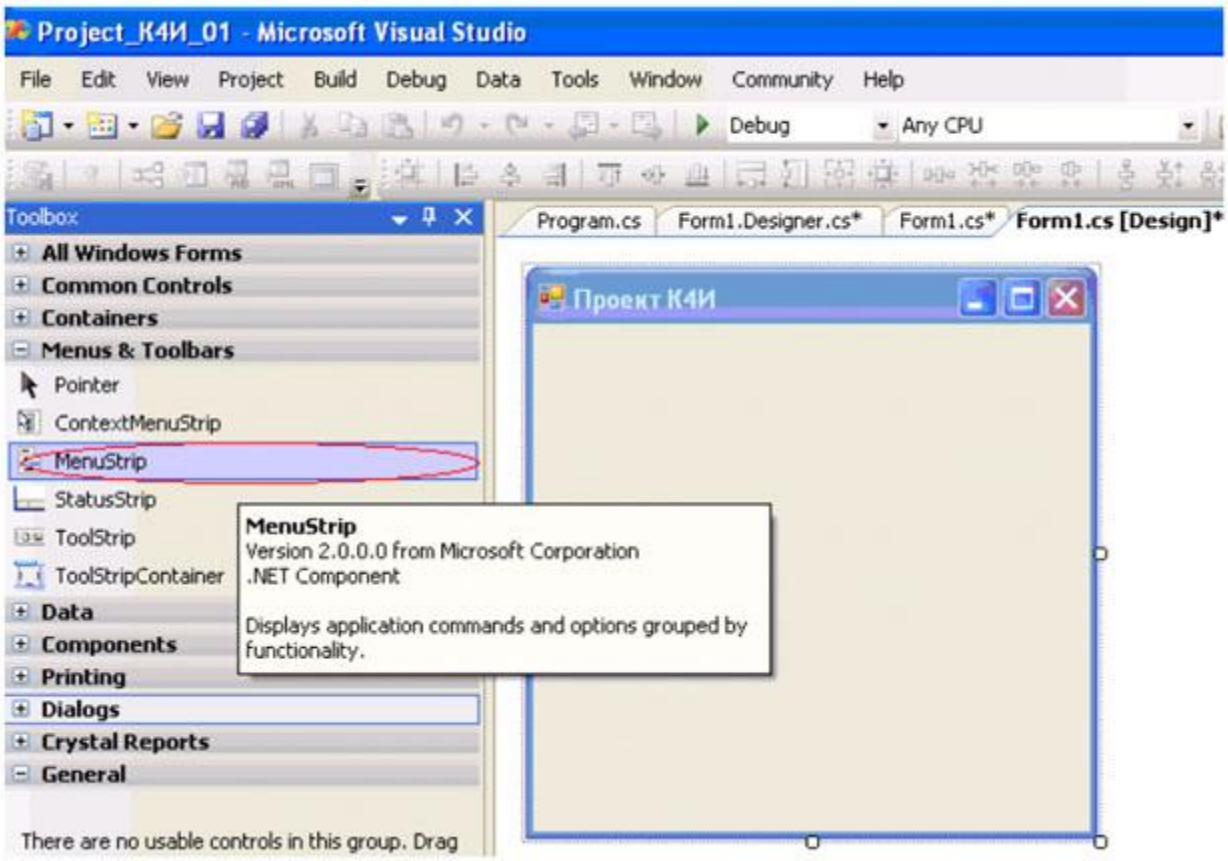

Рис. 2.1. Элемент управления MenuStrip

Элемент управления MenuStrip позволит вводить текст *меню* непосредственно в элементы *меню*. То, что должно получиться, представлено на рисунке 2.2.

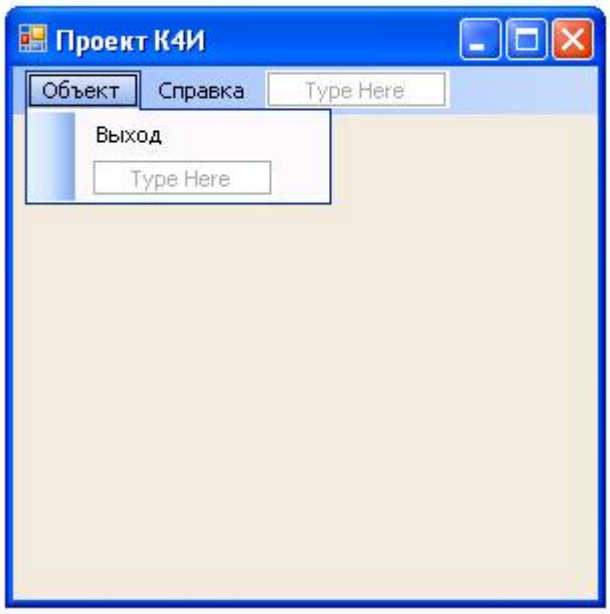

Рис. 2.2. Простое меню на форме

При помощи графических средств можно настроить свойства любого элемента *меню*. Для пункта *меню "Объект"* зададим свойство*Name* равным objektToolStripMenuItem, для пункта *меню "Выход"*) - exitToolStripMenuItem, а для пункта *меню "Справка"* - HelpToolStripMenuItem.
При двойном щелчке на

пункте *меню "Выход"* (*объект* exitToolStripMenuItem) *Visual Studio* автоматически сгенерирует оболочку для обработчика события Click и перейдет в окно кода, в котором нам будет предложено создать логику метода (в нашем случаеexitToolStripMenuItem_Click):

```
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
   {
// Здесь мы определяем реакцию на выбор пользователем
// пункта меню
     }
```

На вкладке *Свойства (Properties)* при выводе окна событий, нажать кнопку  событию Click будет соответствовать методmenuItemExit_Click (рисунок 2.3).

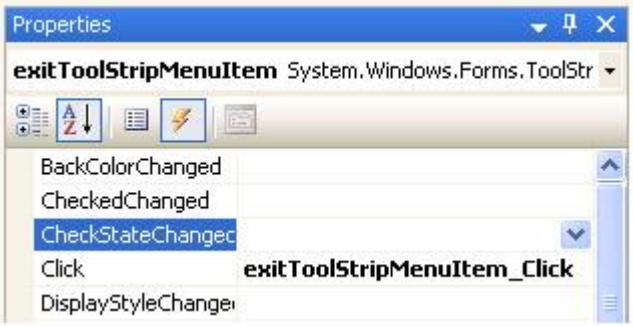

Рис. 2.3. Событие Click и обработчик события exitToolStripMenuItem_Click

Для корректного завершения приложения написать код для обработчика exitToolStripMenuItem_Click. Это можно сделать с помощью метода Exit класса Application:

```
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
  {
   Application.Exit();
  }
```

Для тестирования созданного *меню* создадим обработчик для пункта *меню* "*Объект*", который будет сообщать, что выбран именно этот *пункт меню*.

```
private void objektToolStripMenuItem_Click(object sender, EventArgs e)
  {
   MessageBox.Show("Пункт меню Объект");
  }
```

При создании *меню* графическими средствами *Visual Studio* автоматически внесет необходимые изменения в служебный методInitializeComponent и добавит переменные-члены, представляющие созданные элементы *меню*.

Задание на лабораторную работу

1. Изучить теоретический материал.
2. Создать главное меню, включающее следующие пункты: "Объект", "Справочник", "Справка".
3. Для пункта "Объект" создать следующие подпункты: "Сотрудник", "Клиент", "Договор", "Поручение", "Сделка", "Выход".
4. Для пункта "Справочник" создать следующие подпункты: "Должность", "Страна", "Регион", "Город", "ИМНС".
5. Для пункта "Справка" создать подпункт - "О программе"
6. Протестировать работу приложения.

Лабораторная работа №3

Создание элементов управления

Цель работы: Изучить основные *элементы управления Windows* -форм, их свойства и методы, а также получить практические навыки в разработке *Windows* -форм с элементами контроля

Общие сведения

Классы, представляющие графические *элементы управления*, находятся в пространстве имен *System.Windows.Forms*. С их помощью обеспечивается *реакция* на действия пользователя в приложении *Windows Forms*. Классы элементов управления связаны между собой достаточно сложными отношениями наследования. Общая схема таких отношений представлена на рисунке 3.1.

Класс Control, как общий предок, обеспечивает все *производные* классы общим набором важнейших возможностей. В числе этих возможностей можно перечислить события мыши и клавиатуры, физические размеры и местонахождение элемента управления (свойства Height, Width, Left, Right, Location), установку цвета фона и цвета переднего плана, выбор шрифта и т.п.

При создании приложения можно добавить *элементы управления* на форму при помощи графических средств *Visual Studio*. Обычно достаточно выбрать нужный элемент управления в окне *ToolBox* и поместить его на форме. *Visual Studio* автоматически сгенерирует нужный код для формы. После этого можно изменить название элемента управления на более содержательное (например, вместоbutton1, предлагаемого по умолчанию, - buttonPrimer) *Visual Studio* позволяет не только размещать на форме *элементы управления*, но и настраивать их свойства. Для этого достаточно щелкнуть на элементе управления правой кнопкой мыши и в контекстном *меню* выбрать *Properties* (Свойства).

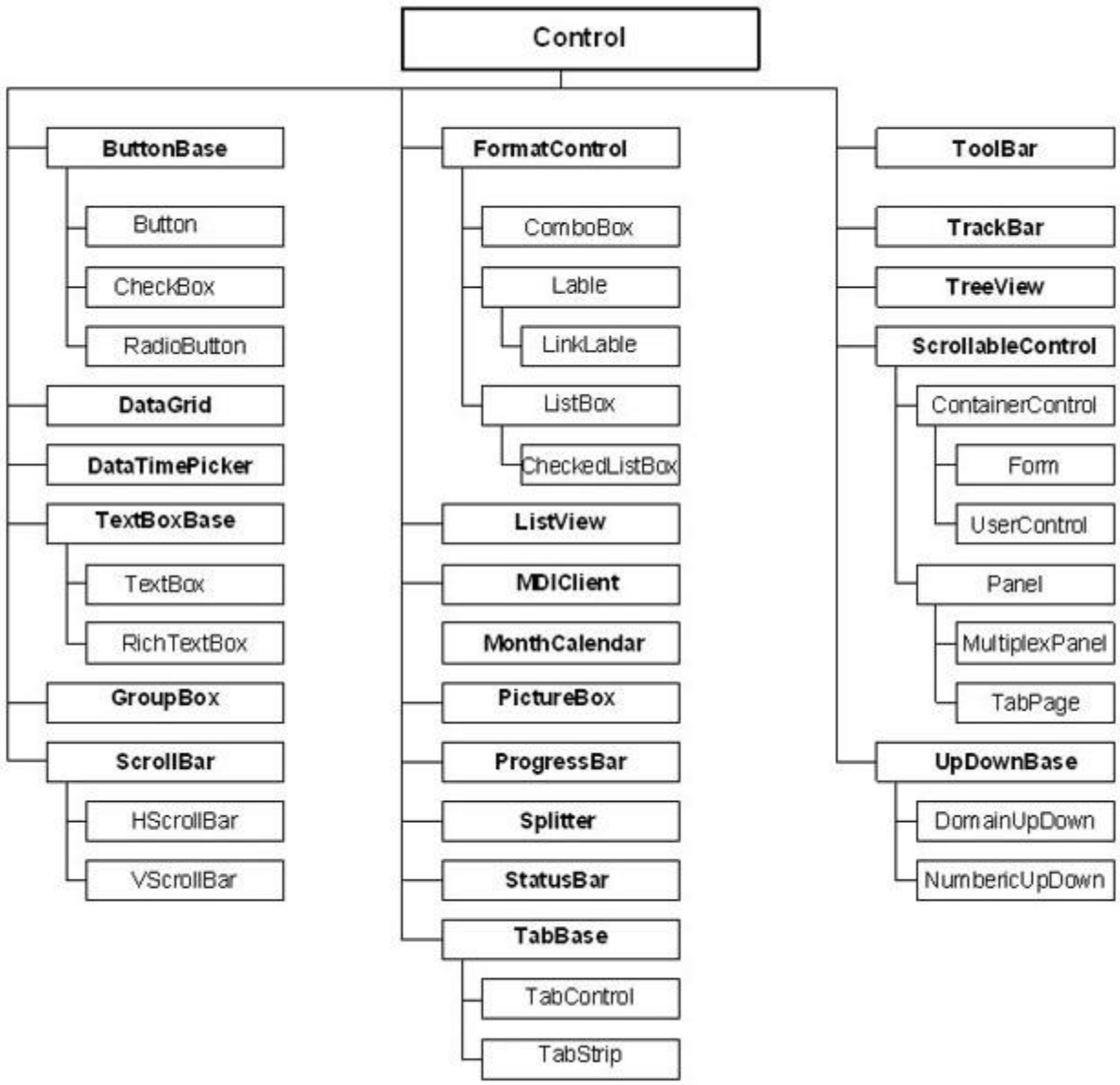

Рис. 3.1. Иерархия элементов управления Windows Forms

Все изменения, которые необходимо произвести в открывшемся окне (рисунок 3.2), будут добавлены в код методаInitializeComponents().

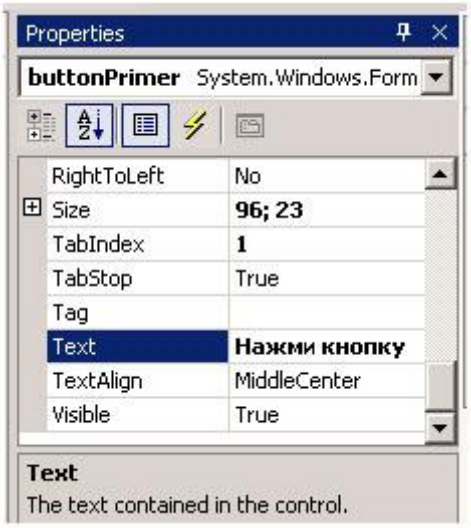

Рис. 3.2. Настройка элементов управления средствами Visual Studio

То же самое окно позволяет настроить не только свойства данного элемента управления, но и обработку событий этого элемента. Перейти в *список событий* можно при помощи кнопки  в закладке *Properties* (рисунок 3.3). Можно выбрать в списке нужное событие и рядом с ним сделать *двойной щелчок* или ввести имя метода или выбрать метод из списка.

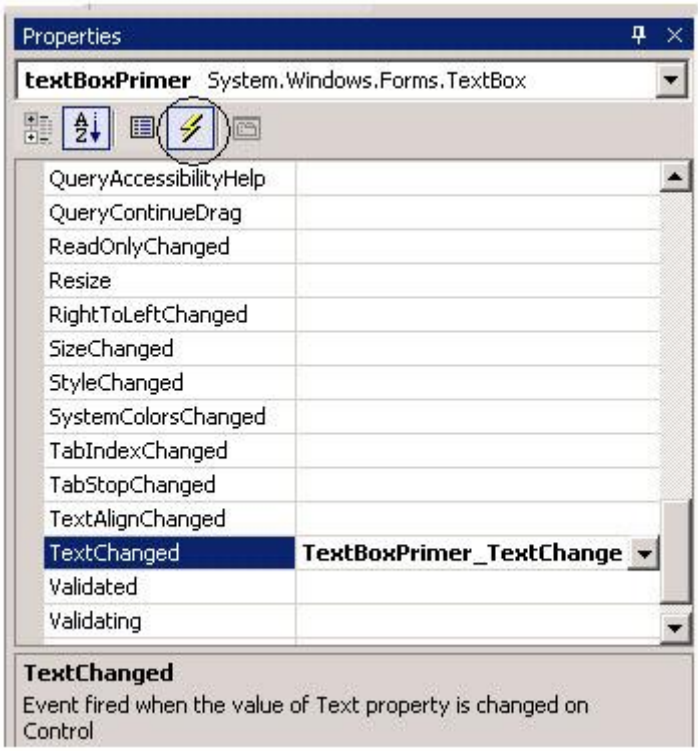

Рис. 3.3. Настройка обработчиков событий

После задания имени метода или двойного щелчка *Visual Studio* сгенерирует заготовку для обработчика события.

Рассмотрим основные *элементы управления Windows* -форм.

Элемент управления TextBox

Элемент управления TextBox (текстовое окно) предназначен для хранения текста (одной или нескольких строк). При желании текст в TextBox может быть настроен как "только для чтения", а в правой и нижней части можно поместить полосы прокрутки.

Класс TextBox происходит непосредственно от класса TextBoxBase, обеспечивает общими возможностями как TextBox, так иRichTextBox. Свойства, определенные в TextBoxBase, представлены в таблице 3.1.

Таблица 3.1. Свойства TextBoxBase

Свойство	**Назначение**
AcceptsTab	Определяет, что будет производиться при нажатии на клавишу *Tab*: вставка символа табуляции в само поле или переход к другому элементу управления
AutoSize	Определяет, будет ли элемент управления автоматически изменять размер при изменении шрифта на нем
BackColor,	Позволяют получить или установить значение цвета фона и переднего

ForeColor	плана
HideSelection	Позволяет получить или установить значение, определяющее, будет ли текст в TextBox оставаться выделенным после того, как этот элемент управления будет выведен из фокуса
MaxLength	Определяет максимальное количество символов, которое можно будет ввести в TextBox
Modified	Позволяет получить или установить значение, определяющее, был ли текст в TextBox изменен пользователем
Multiline	Указывает, может ли TextBox содержать несколько строк текста
Readonly	Помечает TextBox как "только для чтения"
SelectedText, SelectionLength	Содержат выделенный текст (или определенное количество символов) в TextBox
SelectionStart	Позволяет получить начало выделенного текста в TextBox
Wordwrap	Определяет, будет ли текст в TextBox автоматически переноситься на новую строку при достижении предельной длины строки

В TextBoxBase также определено множество методов: для работы с буфером обмена (Cut, Copy и Paste), отменой ввода (Undo) и прочими возможностями редактирования (Clear, AppendText и т. п.).

Из всех событий, определенных в TextBoxBase, наибольший интерес представляет событие TextChange. Это событие происходит при изменении текста в объекте класса, производном от TextBoxBase. Например, его можно использовать для *проверки допустимости* вводимых пользователем символов (например, предположим, что пользователь должен вводить в поле только цифры или, наоборот, только буквы).

Свойства, унаследованные от Control и от TextBoxBase, определяют большую часть возможностей TextBox. Свойств, определенных непосредственно в классе TextBox, не так уж и много. Они представлены в таблице 3.2.

Таблица 3.2. Свойства, определенные в классе TextBox

Свойство	**Назначение**
AcceptsReturn	Позволяет определить, что происходит, когда пользователь при вводе текста нажал на Enter. Варианта два: либо в TextBox начинается новая строка текста, либо активизируется кнопка по умолчанию на форме
CharacterCasing	Позволяет получить или установить значение, определяющее, будет ли изменяться регистр вводимых пользователем символов
PasswordChar	Позволяет выбрать символ, используемый для отображения вводимых пользователем данных (в поле для ввода пароля)
ScrollBars	Позволяет получить или установить значение, определяющее, будут ли в TextBox с несколькими строками присутствовать полосы прокрутки
TextAlign	Позволяет определить выравнивание текста в TextBox (используются значения из перечисленияHorizontalAlignment)

Значения перечисления HorizontalAlignment представлены в таблице 3.3.

Таблица 3.3. Значения перечисления HorizontalAlignment

Значение	**Описание**
Center	Выравнивание по центру
Left	Выравнивание по левому краю

Right	Выравнивание по правому краю

Класс Button

Кнопка (button) - это самый простой из всех элементов управления и при этом наиболее часто используемый. Можно сказать, что кнопка - это возможность принять ввод (щелчок кнопкой мыши или набор на клавиатуре) наиболее простым способом. Непосредственный предок класса System.Windows.FormButton в иерархии классов *.NET* - это класс ButtonBase, обеспечивающий общие возможности для целой группы производных от него элементов управления (таких как Button, CheckBox и RadioButton). Некоторые свойства ButtonBase представлены в таблице 3.4.

Таблица 3.4. Свойства ButtonBase

Свойство	**Назначение**
FlatStyle	Позволяет настроить "рельефность" кнопки. Используются значения из перечисления FlatStyle
Image	Позволяет задать изображение, которое будет выводиться на кнопке (при этом можно указать точное местонахождение изображения). Фоновый рисунок лучше настраивать при помощи свойства BackgroundImage, определенного в базовом классе Control
ImageAlign	Позволяет определить выравнивание изображения, размещенного на кнопке. Используются значения из перечисленияContentAlignment
Imagelndex, ImageList	Эти свойства используются для работы с набором изображений (объектом ImageList), выводимых на кнопке
IsDefault	Определяет, будет ли эта кнопка являться кнопкой по умолчанию (то есть срабатывать при нажатии на *Enter*)
TextAlign	Позволяет получить или установить выравнивание текста на кнопке. Также используются значения из перечисленияContentAlignment

Сам класс Button не определяет каких-либо дополнительных возможностей помимо унаследованных от ButtonBase, за единственным, но существенным исключением свойства DialogResult. Это свойство позволяет возвращать значение при закрытии диалогового окна, например, при нажатии кнопок *OK* или *Cancel* (Отменить).

В подавляющем большинстве случаев выравнивание текста, размещенного на кнопке, производится по центру, так что текст будет размещен строго посередине кнопки. Однако если нам по каким-то причинам необходимо использовать другой стиль выравнивания, в нашем распоряжении - свойство TextAlign, определенное в классе ButtonBase. Для TextAlign используются значения из перечисления ContentAlignment (таблица 3.5). Значения из того же перечисления используются и для определения положения изображения на кнопке.

Таблица 3.5. Значения перечисления ContentAlignment

Значение	**Описание (выравнивание)**
BottomCenter	По нижнему краю кнопки, относительно боковых краев - посередине
BottomLeft	Но нижнему краю кнопки, слева
BottomRight	По нижнему краю кнопки, справа
MiddleCenter	По центру кнопки
MiddleLeft	Относительно верхнего и нижнего краев - по центру, относительно боковых краев - слева
MiddleRight	Относительно верхнего и нижнего краев - по центру, относительно боковых краев - справа
TopCenter	По верхнему краю кнопки, относительно боковых краев - посередине

TopLeft	По верхнему краю кнопки, слева
TopRight	По верхнему краю кнопки, справа

Флажки

Для флажка (тип CheckBox) предусмотрено три возможных состояния. Как и тип Button, CheckBox наследует большую часть своих возможностей от базовых классов Control и ButtonBase. Однако в этом классе существуют и свои собственные члены, обеспечивающие дополнительные уникальные возможности. Наиболее важные свойства CheckBox представлены в таблице 3.6.

Таблица 7.6. Свойства класса CheckBox

Свойство	**Назначение**
Appearance	Настраивает вид флажка. Для этого свойства используются значения из перечисления Appearance
AutoCheck	Позволяет получить или установить значение, определяющее, будут ли значения Checked и CheckState, а также внешний вид флажка автоматически изменяться при щелчке на нем
CheckAlign	Позволяет установить горизонтальное и вертикальное выравнивание собственно флажка (квадратика) в элементе управления CheckBox. Используются значения из перечисления ContentAlignment
Checked	Возвращает значение типа bool, представляющее текущее состояние флажка (выбран или не выбран) Если для свойства ThreeState установлено значение true, то свойство Checked будет возвращать true как для явно выбранного флажка, так и для того флажка, для которого установлено значение "не определено" (*indeterminate*)
CheckState	Позволяет получить или установить значение флажка (установлен - не установлен - не определено), используя нетtrue и false, как в Checked, а три значения из перечисления CheckState. Обычно используется, если свойство ThreeState для флажка имеет значение true (то есть он допускает три значения).
ThreeState	Определяет, будут ли для флажка использоваться три значения (из перечисления CheckState) или только два

Возможные состояния флажка (*Indeterminate* можно использовать только тогда, когда для свойства ThreeState установлено значение true) представлены в таблице 3.7.

Таблица 3.7. Значения перечисления CheckState

Значение	**Описание**
Checked	Флажок установлен
Indeterminate	Значение не определено (обычно флажок выглядит как "серый", затененный)
Unchecked	Флажок снят

Состояние "значение не определено" (indeterminate) может быть установлено, например, для верхнего элемента иерархии, в которой для одной части подчиненных элементов флажок установлен, а для другой - снят. Переключатели и группирующие рамки

Тип RadioButton (переключатель) можно воспринимать, как несколько видоизмененный флажок при этом сходство между этими типами подчеркивается почти полным совпадением наборов членов. Между типами RadioButton и CheckBox существуют лишь два важных различия: в RadioButton предусмотрено событие CheckedChanged (возникающее при

изменении значения Checked), а кроме того, RadioButton не поддерживает свойство ThreeState и не может принимать состояние *Indeterminate* (не определено).

Переключатели всегда используются в группах, которые рассматриваются как некое единое целое. Внутри группы переключателей одновременно может быть выбран только один переключатель. Для группировки переключателей в группы используется типGroupBox.

И флажок (CheckBox), и переключатель (RadioButton) поддерживают свойство Checked, при помощи которого очень удобно получать информацию о состоянии соответственно флажка и переключателя. Однако если есть необходимость задействовать дополнительное третье состояние флажка (не определено - *Indeterminate*), то придется вместо Checkedиспользовать свойство CheckState и значения из одноименного перечисления CheckState.

Элемент управления CheckedListBox

Типы Button, CheckBox и RadioButton являются производными от ButtonBase, и их можно определить как некие разновидности кнопок. К членам семейства списков относятся CheckedListBox (список с флажками), ListBox (список) иComboBox (комбинированный список).

Элемент управления CheckedListBox (список с флажками) позволяет помещать обычные флажки внутри поля с полосами прокрутки.

Кроме того, в элементе управления CheckedListBox предусмотрена возможность использования нескольких столбцов. Для этого достаточно установить значение true для свойства MultiColumn.

CheckedListBox наследует большинство своих возможностей от типа ListBox. То же самое справедливо и в отношении классаComboBox. Наиболее важные свойства System.Windows.Forms.ListBox представлены в таблице 3.8.

Таблица 3.8. Свойства класса ListBox

Свойство	**Назначение**
ScrollAlwaysVisible	Определяет, будет ли полоса прокрутки выводиться всегда
Selectedlndex	Индекс выделенного в настоящий момент элемента в списке (если такой имеется). Если ни один элемент не выделен, то возвращается значение -1
SelectedIndices	Набор индексов выделенных в настоящий момент элементов в списке. Если не выделен ни один элемент, то возвращается пустой набор
SelectedItem	Значение выделенного в настоящий момент элемента. Если ни один из элементов не выделен, то возвращается null
SelectedItems	Возвращает коллекцию значений выделенных элементов (для списков, в которых допускается выбор нескольких значений)
SelectionMode	Определяет число элементов, которые возможно выбрать в списке одновременно. Для этого свойства используются значения из перечисления SelectionMode
Sorted	Определяет, будут ли элементы в списке упорядочены (по алфавиту) или нет
TopIndex	Возвращает индекс первого видимого элемента в списке

Помимо свойств в классе ListBox определены также многочисленные методы. Подавляющее большинство этих методов дублирует возможности, предоставляемые в наше распоряжение свойствами, поэтому мы их рассматривать не будем.

Комбинированные списки

Как и списки (объекты ListBox), комбинированные списки (объекты ComboBox) позволяют пользователю производить выбор из списка заранее определенных элементов. Однако у комбинированных списков есть одно существенное отличие от обычных: пользователь может не только выбрать готовое значение из списка, но и ввести свое собственное. Класс ComboBox наследует большинство своих возможностей от класса ListBox (который, в свою очередь, является производным от Control), однако в нем предусмотрены и собственные важные свойства, представленные в таблице 3.9.

Таблица 3.9. Свойства ComboBox

Свойство	**Назначение**
DroppedDown	"Раскрывающийся вниз": определяет, будет ли список ниспадающим
MaxDropDownltems	Определяет максимальное количество элементов, которое будет показано в нижней части ниспадающего списка. Допустимые значения - от 1 до 100
MaxLength	Определяет максимальную длину текста, который пользователь может ввести в ComboBox
Selectedlndex	Определяет индекс выделенного элемента ComboBox. Если ни один элемент не выделен, возвращается значение -1
Selectedltem	Возвращает ссылку на объект выделенного элемента ComboBox
SelectedText	Возвращает выделенный текст в поле редактирования ComboBox
SelectionLength	Определяет длину (в символах) выделенного текста в поле редактирования ComboBox
Style	Позволяет получить или установить стиль ComboBox. Для этого свойства используются значения из перечисления ComboBoxStyle
Text	Позволяет получить доступ к тексту в поле редактирования. При работе с ComboBox это унаследованное свойство используется чаще всех остальных

Стиль для ComboBox можно настроить при помощи свойства Style, для которого используются значения из перечисленияComboBoxStyle (таблица 3.10).

Таблица 3.10. Значения перечисления ComboBoxStyle

Значение	**Описание**
DropDown	Пользователь может вводить значения в поле редактирования. Для отображения списка пользователь должен нажать на кнопку со стрелкой, направленной вниз (Arrow Button)
DropDownList	Пользователь не может вводить значения в поле редактирования. Для отображения списка пользователь должен нажать на кнопку со стрелкой, направленной вниз (Arrow Button)
Simple	Пользователь может вводить значения в поле редактирования. Список значений виден всегда

Порядок перехода по Tab

Если на форме размещено несколько элементов управления, то пользователи обычно ожидают, что между ними можно будет перемещаться с помощью клавиши *Tab*. Часто бывает необходимо после размещения элементов управления настроить порядок перехода между ними. Для этого используются два свойства (унаследованные от базового класса Control и поэтому общие для всех элементов управления): TabStop и Tablndex. Для свойства TabStop используются только два значения: true и false. Если для TabStop установлено значение true, то к этому элементу управления можно будет

добраться с помощью клавиши *Tab*. Если же установлено значение false, то участвовать в переходах по *Tab* этот элемент управления не будет. Если элемент управления TabStop имеет значение true, то очередность перехода можно настроить с помощью свойства Tablndex:

В *Visual Studio.NET* предусмотрено средство, при помощи которого можно быстро настроить порядок перехода для элементов управления на форме. Это средство называется *Tab Order Wizard* и оно доступно из меню *View (View > Tab Order)*. Чтобы изменить значения TabIndex для каждого элемента управления, достаточно просто щелкать мышью на элементах управления в выбранном нами порядке перехода. Для элементов управления, помещенных в группирующую рамку, *Tab Order Wizard* создает отдельную последовательность перехода.

Элемент управления MonthCalendar

В пространстве имен *System.Windows.Forms* предусмотрен элемент управления, при помощи которого пользователь может выбрать дату или диапазон дат, используя дружественный и удобный интерфейс. Это элемент управления MonthCalendar.

Наиболее важные свойства MonthCalendar представлены в табл. 3.11.

Таблица 3.11. Свойства MonthCalendar

Свойство	**Назначение**
BoldedDates	Массив объектов DateTime, выделенных подсветкой
CalendarDimensions	Определяет количество выводимых строк и столбцов
FirstDayOfWeek	Определяет, с какого дня будет начинаться неделя в MonthCalendar
MaxDate	Самая поздняя дата, которую разрешается выбрать пользователю (по умолчанию ограничений нет)
MaxSelectionCount	Максимальное количество дат, которое одновременно может выбрать пользователь
MinDate	Самая ранняя дата, которую разрешается выбрать пользователю (по умолчанию ограничений нет)
MonthlyBoldedDates	Массив выделенных подсветкой объектов DateTime для месяца
SelectionRange	Диапазон выделенных объектов
SelectionEnd	Самая поздняя дата в диапазоне выделенных объектов
SelectionStart	Самая ранняя дата в диапазоне выделенных объектов
ShowToday	Определяет, будет ли MonthCalendar выводить информацию о текущей дате
ShowTodayCircle	Определяет, будет ли MonthCalendar выводить информацию о текущей дате в нижней части и выделять ее в календаре обводкой
ShowWeekNumbers	Определяет, будет ли MonthCalendar отображать номера недель справа от каждой строки
TodayDate	Дата, которая будет считаться MonthCalendar сегодняшней. По умолчанию TodayDate - это системная дата на момент создания объекта MonthCalendar
TodayDateSet	Определяет, можно ли пользователю по своему усмотрению выбирать сегодняшнюю дату. Если для этого свойства установлено значение true, пользователь может выбрать в качестве сегодняшней (TodayDate) любое число

По умолчанию всегда выделяется (и подсветкой, и обводкой) текущая дата. Пользователь может выбрать другую дату - в этом и есть смысл графического интерфейса MonthCalendar.

Можно получить дату, выбранную пользователем в MonthCalendar, при помощи свойства SelectionStart. Это свойство возвращает ссылку на объект DateTime, которая хранится в специальной переменной (d) При помощи набора свойств типаDateTime можно извлечь всю необходимую информацию в нужном нам формате.

При помощи свойств Month, Day и Year можно извлечь из объектов DateTime нужную информацию и сформировали текстовые строки. Это вполне допустимый подход. Дело в том, что дату в необходимом текстовом формате проще получить изDateTime при помощи специальных "форматирующих" свойств самих объектов DateTime. Набор таких свойств (и некоторые методы) представлен в таблице 3.12.

Таблица 3.12. Члены класса DateTime

Член	**Назначение**
Date	Позволяет получить информацию о дате (дата всегда отсчитывается от полуночи)
Day, Month, Year	Позволяют получить соответственно день, месяц и число из текущего объектаDateTime
DayOfWeek	Возвращает день недели для объекта DateTime
DayOfYear	Возвращает номер дня в году для объекта DateTime
Hour, Minute, Second, *Millisecond*	Возвращают информацию о часе, минутах, секундах и миллисекундах для объекта DateTime
MaxValue, *MinValue*	Возвращают минимальное и максимальное значения для DateTime
Now, Today	Эти два статических свойства типа DateTime позволяют получить информацию о текущей дате и времени (Now) или только о текущей дате (Today)
Ticks	Позволяет получить счетчик "тиков" (с интервалом в 100 наносекунд) для объекта DateTime
ToLongDateString( ), ToLongTimeString( ), ToShortDateString( ), ToShortTimeString( )	Преобразуют текущее значение объекта DateTime в разные виды текстового представления

При помощи вышеперечисленных членов можно значительно упростить вывод текстовой информации о дате.

Элемент управления Panel

Назначение элемента управления Panel (панель) - с его помощью можно объединить прочие элементы управления на форме.Panel происходит от базового класса ScrollableControl и поддерживает полосы прокрутки.

Элементы управления Panel обычно используются для экономии пространства на форме. Например, если элементы управления, которые планируем разместить на форме, на ней не умещаются, то можно поместить их внутрь Panel и установить для свойстваAutoScrol1 объекта Panel значение true. В результате пользователь получит возможность доступа к "не вмещающимся" элементам управления с помощью полос прокрутки.

Всплывающие подсказки (ToolTips)

Большинство приложений с современным пользовательским интерфейсом поддерживают всплывающие подсказки. В приложениях.*NET* эта возможность реализуется при помощи

типа System.Windows.Forms.ToolTip. ToolTip (всплывающие подсказки) - это небольшие окна с текстом, появляющиеся при наведении указателя мыши на элемент управления на форме. Наиболее важные члены класса ToolTip представлены в таблице 3.13.

Таблица 3.13. Члены класса ToolTip

Член	**Назначение**
Active	Определяет, будет ли всплывающая подсказка активной. Возможность отключить всплывающие подсказки может быть полезной, например, если в приложении предусмотрено два варианта интерфейса: для обычных и для опытных пользователей
AutomaticDelay	Позволяет получить или установить время задержки (в миллисекундах) при появлении подсказки
AutoPopDelay	Время (в миллисекундах), в течение которого подсказка остается видимой, если указатель мыши неподвижен и находится в области, занимаемой соответствующим элементом управления. По умолчанию это значение равно 10 значениям AutomaticDelay
GetTooltip( )	Возвращает текст подсказки
InitialDelay	Время (в миллисекундах), в течение которого указатель должен оставаться неподвижным в соответствующей области для появления подсказки. Значение по умолчанию равно значению AutomaticDelay
ReshowDelay	Время (в миллисекундах), в течение которого появится другая подсказка при перемещении указателя мыши от одного элемента управления к другому. По умолчанию это значение равно 1/5 от значения AutomaicDelay
SetToolTip( )	Ассоциирует подсказку с элементом управления

Для того чтобы настроить использование всплывающих подсказок для элементов управления можно сделать это с помощью графических средств *Visual Studio*.

Первое, что необходимо сделать - добавить на форму объект *ToolTip*, выбрав его в *ToolBox*. Затем можно указать текст всплывающей подсказки для любого элемента управления на форме (в том числе и для самой формы) из окна свойств данного элемента.

Проектирование элементов управления формы FormEmployee

Для разработки проекта приложения форма *FormEmployee* должна содержать *элементы управления*, в соответствии с видом, приведенном на рисунке 3.4 .

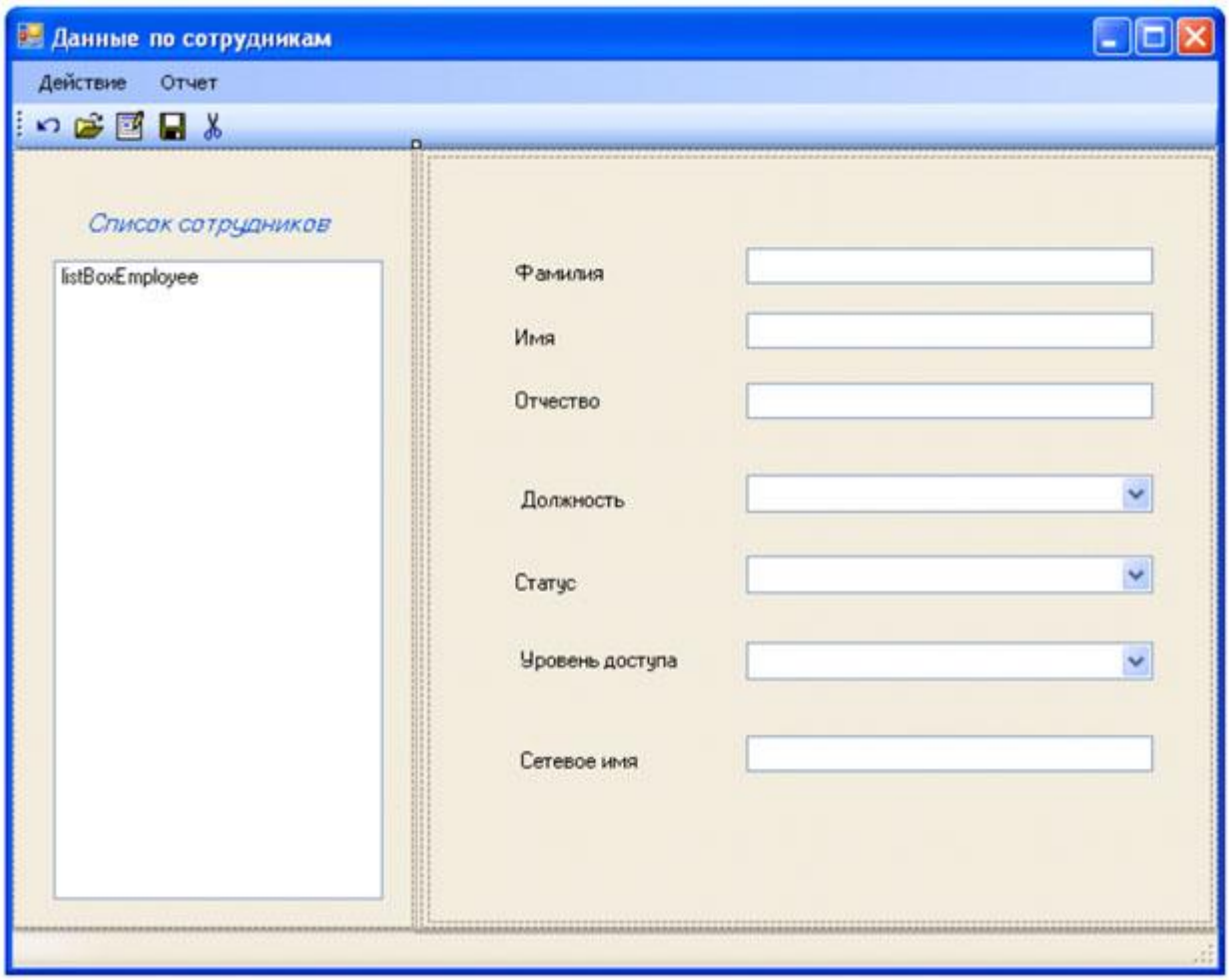

Рис. 3.4. Вид экранной формы FormEmployee

На данной форме необходимо сформировать *элементы управления*, приведенные в таблице 3.14.

Таблица 3.14. Элементы управления формы FormEmployee

Элемент контроля	**Имя**	**Свойство Text/Items**	**Назначение**
SplitContainer	splitContainerEmployee		Две панели с разделителем
Label	labelListEmployee	Список сотрудников	Надпись
listBox	listBoxEmployee		Список сотрудников
Label	labelSurname	Фамилия	Надпись
Label	labelName	Имя	Надпись
Label	labelPatronymic	Отчество	Надпись
Label	labelJobRole	Должность	Надпись
Label	labelStatus	Статус	Надпись
Label	labelAccess	Уровень доступа	Надпись
label	labelNetName	Сетевое имя	Надпись
textBox	textBoxNetName		Сетевое имя
textBox	textBoxSurname		Фамилия
textBox	textBoxName		Имя
textBox	textBoxPatronymic		Отчество
comboBox	comboBoxJobRole		Должность

comboBox	comboBoxStatus	Активен, выходной, в отпуске, болеет, не работает, помечен как удаленный	Статус
comboBox	comboBoxAccess	Оператор, старший оператор, начальник смены, администратор, аналитик	Уровень доступа
menuItem	menuItemAction	Действие	Пункт меню "Редактировать"
menuItem	menuItemUndo	Отменить	Подпункт меню "Отменить"
menuItem	menuItemNew	Создать	Подпункт меню "Новый"
menuItem	menuItemEdit	Изменить	Подпункт меню "Изменить"
menuItem	menuItemSave	Сохранить	Подпункт меню "Сохранить"
menuItem	menuItem	Удалить	Подпункт меню "Удалить"
menuItem	menuItemReport	Отчет	Пункт меню "Отчет"
menuItem	menuItemReport1	По сотруднику	Подпункт меню "Отчет по сотруднику"
menuItem	menuItemReport2	По всем сотрудникам	Подпункт меню "Отчет по всем сотрудникам"

Вначале создайте на форме элемент *SplitContainer* (рисунок 3.5). На панели 1 создайте *элементы управления* labelListEmployee иlistBoxEmployee (рисунок 3.6), а остальные *элементы управления*, приведенные в таблице 3.14, - на панели 2 (рисунок 3.4).

После создания на форме *FormEmployee* элементов управления в соответствии с таблицей 3.14 необходимо настроить порядок перехода между ними при нажатии клавиши *Tab*.

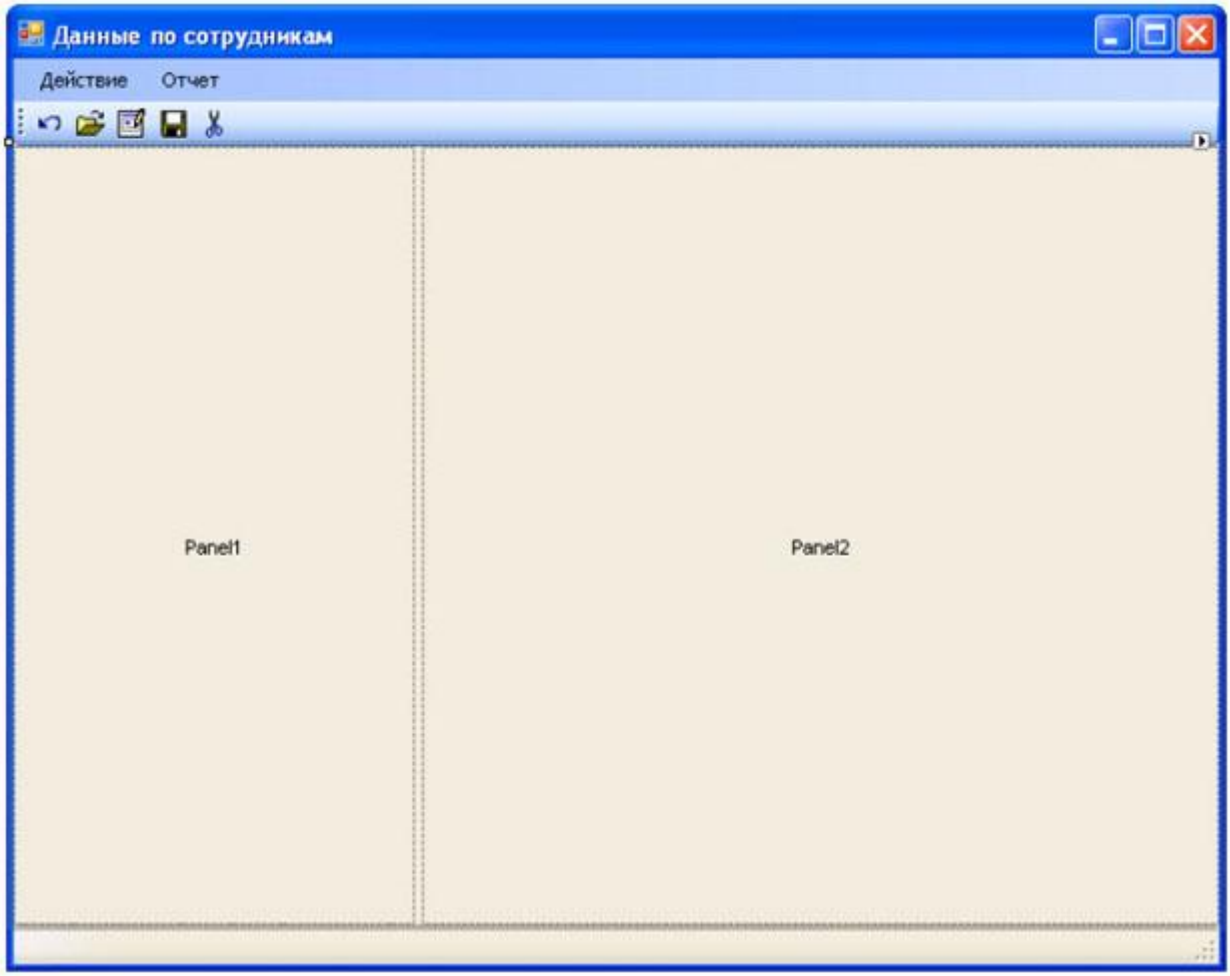

Рис. 3.5. Создание панелей на форме FormEmployee

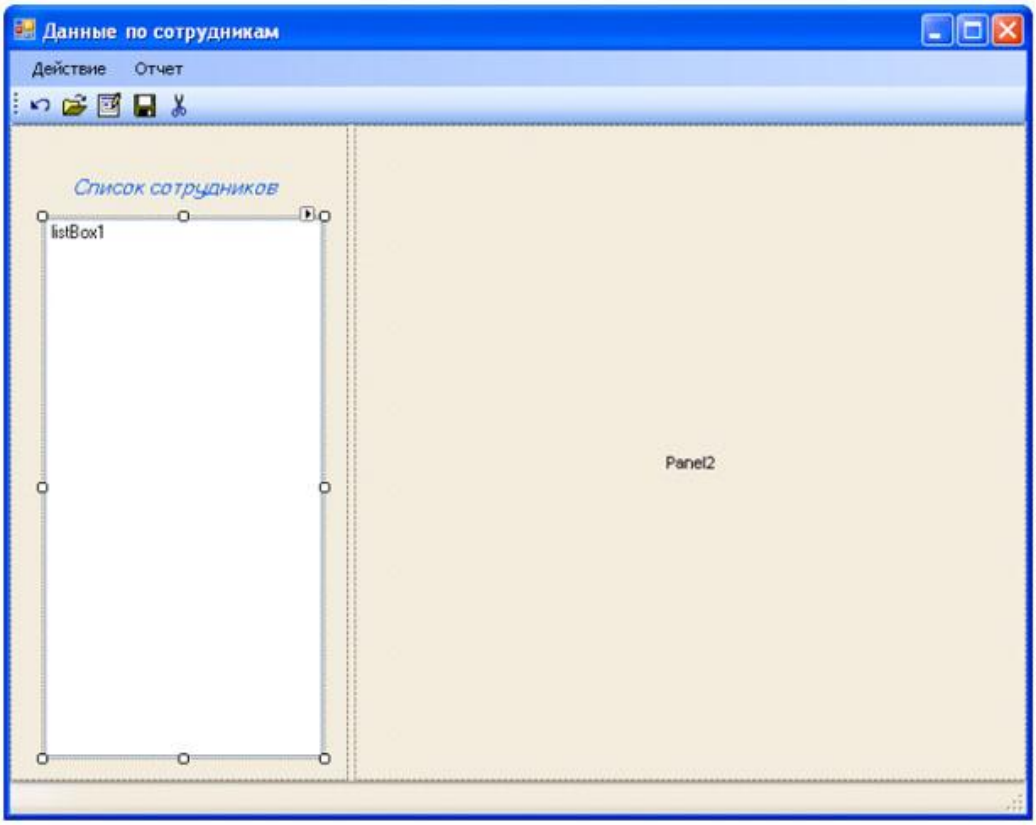

Рис. 3.6. Формирование элементов управления на панели 1 формы FormEmployee

Для этого необходимо задать последовательные номера свойству TabIndex элементов управления (в разрабатываемой форме это необходимо сделать для элементов управления TextBox и ComboBox) из окна *Properties* (рисунок 3.7) или вызвать мастер Tab Order Wizard из *меню View/Tab Order* (рисунок 3.8). Задание последовательности значений свойству TabIndex производится щелчком мыши на элементах управления в заданной последовательности.

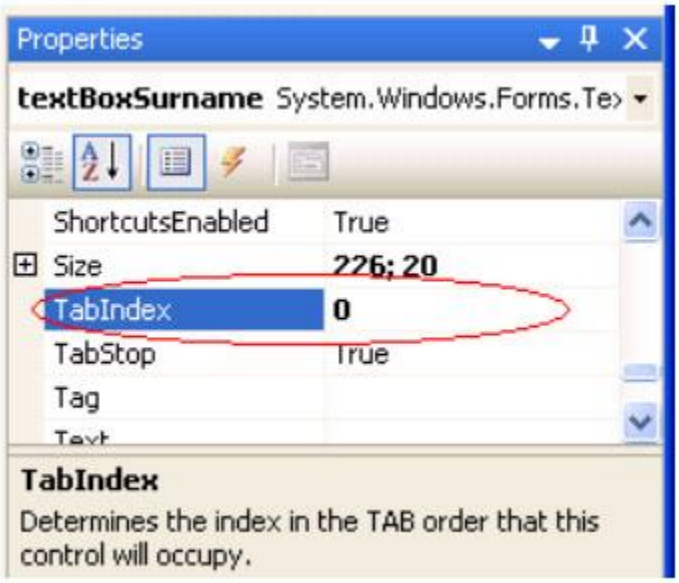

Рис. 3.7. Задание свойства TabIndex для элемента контроля

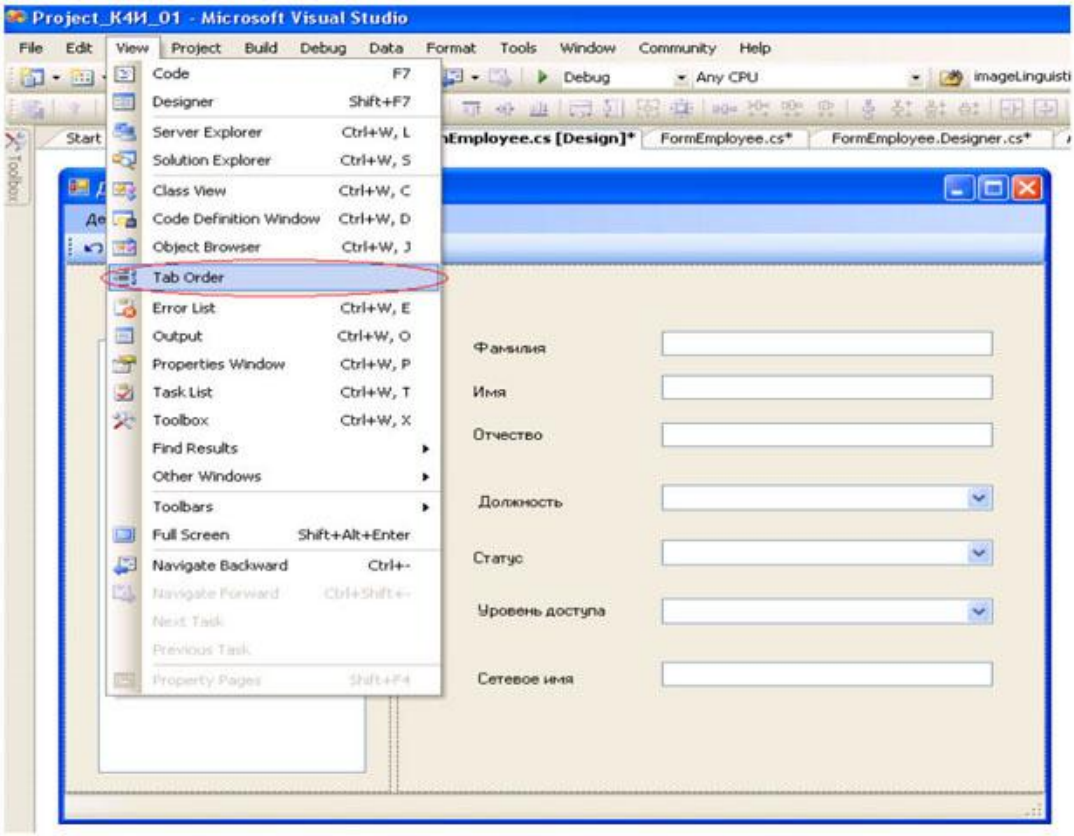

Рис. 3.8. Настройка перехода по элементам управления

Результат настройки порядка перехода между элементами управления при нажатии клавиши *Tab* приведен на рисунке 3.9.

Для работы с формой необходимо создать методы, которые разрешают только просматривать форму (режим просмотра) и редактировать форму (режим редактирования).

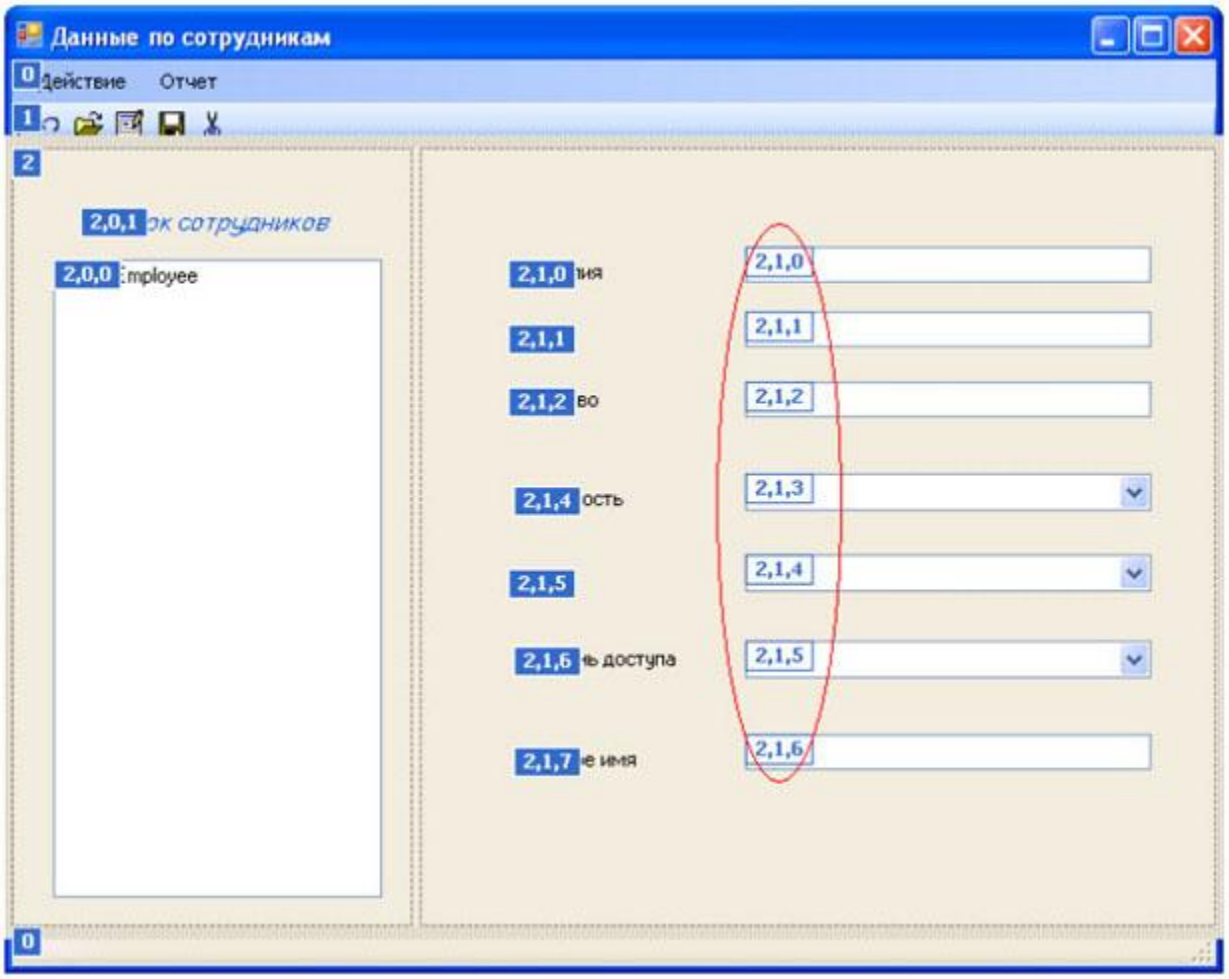

Рис. 3.9. Результат работы мастера Tab Order Wizard

Создадим метод для задания режима просмотра формы DisplayReadOnly. Метод DisplayReadOnly должен быть общедоступным, ничего не должен возвращать и не иметь параметров. Для задания режима просмотра (только для чтения) объекту класса TextBoxнеобходимо свойству ReadOnly присвоить *значение* true, а для объекта класса comboBox - свойству Enabled *значение*false. Код метода DisplayReadOnly представлен далее:

```
public void DisplayReadOnly()
{
   this.textBoxSurname.ReadOnly = true;
   this.textBoxName.ReadOnly = true;
   this.textBoxPatronymic.ReadOnly = true;
   this.textBoxNetName.ReadOnly = true;
   this.comboBoxJobRole.Enabled = false;
   this.comboBoxStatus.Enabled = false;
   this.comboBoxAccess.Enabled = false;
}
```

Аналогичным образом сформируем метод *DisplayEdit*, который задает режим редактирования формы:

```
/// Задание режима редактирования
public void DisplayEdit()
```

```
{
   this.textBoxSurname.ReadOnly = false;
   this.textBoxName.ReadOnly = false;
   this.textBoxPatronymic.ReadOnly = false;
   this.textBoxNetName.ReadOnly = false;
   this.comboBoxJobRole.Enabled = true;
   this.comboBoxStatus.Enabled = true;
   this.comboBoxAccess.Enabled = true;
}
```

Для управления режимом доступности (только для чтения/редактирование) формы *FormEmployee* необходимо методDisplayReadOnly вызывать при первоначальной загрузке формы (событие Load), при создании новых данных по сотруднику и при редактировании данных по сотруднику, а метод *DisplayEdit* - при сохранении данных по сотруднику и при отмене режима редактирования данных.

Проверьте правильность режима управления доступностью элементов управления формы *FormEmployee*.

Анализ кодов методов DisplayReadOnly( ) и *DisplayEdit*( ) показывает, что они могут быть объединены в один метод с параметром. Необходимо самостоятельно написать объединенный метод, получив в результате метод DisplayReadOnly(bool readOnly), в котором параматр readOnly определяет режим редактирования: если readOnly равен true, то режим только для просмотра, если равен false, то - редактирование.

В процессе работы приложения необходимо управлять доступом к пунктам *меню* в соответствии с диаграммой состояний для пунктов *меню* формы *FormEmployee*, приведенной на рисунке 3.10.

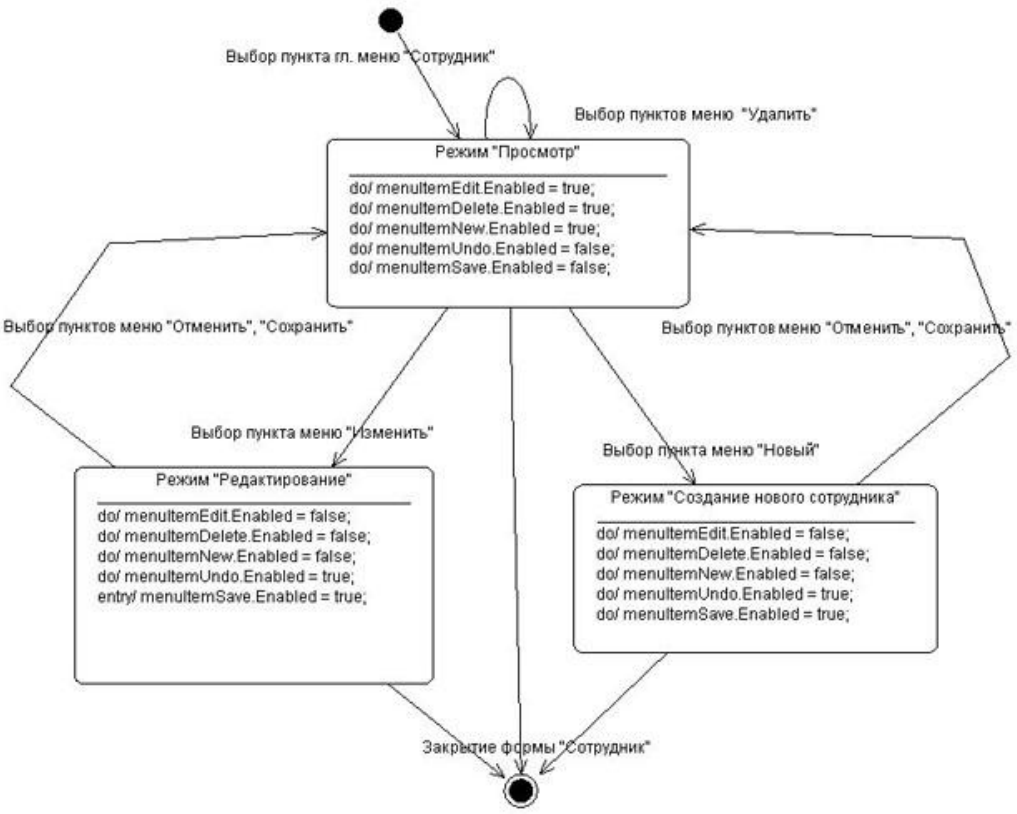

Рис. 3.10. Диаграмма состояний для активности подпунктов меню "Действие"

Диаграмма отображает возможные переходы между тремя режимами: "Просмотр", "Редактирование" и "Создание нового сотрудника".

При выборе в главном *меню* приложения пункта "Сотрудник" *Windows* - форма *FormEmployee* должна перейти в режим "Просмотр", что определяет *доступ* к пунктам *меню* "Создать", "Редактировать", "Удалить" и запрет доступа к подпунктам *меню* "Отменить", "Сохранить".

Если в режиме просмотр выбирается подпункт *меню* "Удалить", то в результате выполнения данной функции режим *Windows*-формы*FormEmployee* не должен измениться, т.е. форма должна остаться в режиме "Просмотр".

Если в режиме просмотр выбирается подпункт *меню* "Изменить", то *Windows*-формы FormEmployee должна перейти в режим "Редактирование". Данный режим предполагает, что разрешается *доступ* к подпунктам *меню* "Отменить", "Сохранить" и запрещается доступа к подпунктам *меню* "Создать", "Редактировать", "Удалить".

Аналогичным образом интерпретируются переходы формы *FormEmployee* из одного режима в другой.

На рисунке 3.10 представлены режимы и переходы для подпунктов главного *меню*. Аналогичные режимы необходимо соблюдать для контекстного *меню* и кнопок панели инструментов.

Для управления доступом к пунктам главного *меню* создайте методы MenuItemEnabled(bool itemEnabled), для контекстного*меню* - MenuItemContextEnabled(bool itemEnabled) и для кнопок панели управления - StripButtonEnabled(bool itemEnabled). Управление доступностью пунктов главного и контекстного *меню* осуществляется через свойство Enabled классаToolStripMenuItem, а кнопок панели управления - через свойство Enabled класса ToolStripButton.

Проверьте правильность режима управления пунктов главного и контекстного *меню*, а также кнопок панели управления формы*FormEmployee*.

С учетом того, что установка режимов просмотра и редактирования экранной формы, а также *управление доступом* к пунктам *меню*должно выполняться при реализации нескольких функций программы целесообразно для избежания дублирования кода все методы управления режимами объединить в один метод DisplayForm.

```
private void DisplayForm(bool mode)
{
   DisplayReadOnly(mode);
   MenuItemEnabled(mode);
   MenuItemContextEnabled(mode);
   StripButtonEnabled(mode);
}
```

Первоначальная установка режима *"Просмотр"* должна проводиться при первоначальной загрузке формы *FormEmployee*.

Задание на лабораторную работу

1. Изучить теоретический материал.
2. Для формы *FormEmployee* создать требуемые элементы контроля.

3. Разработать методы для задания режимов "Просмотр", "Редактирование" для элементов контроля.
4. Разработать методы для задания режимов "Просмотр", "Редактирование" для управления активностью пунктов главного меню формы, контекстного меню и кнопок панели инструментов.
5. Сформировать обработчик события Load.
6. Протестировать программу.

Лабораторная работа №4

Подготовка ADO.NET к работе в приложении

Цель работы: Изучить назначение и основные способы создания объектов *ADO.NET* при помощи *Visual Studio IDE*

Общие сведения

В платформе *.NET* определено множество типов (организованных в соответствующие пространства имен) для взаимодействия с локальными и удаленными хранилищами данных. Общее название пространств имен с этими типами - *ADO.NET*.

ADO.NET - это новая технология доступа к базам данных, специально оптимизированная для нужд построения рассоединенных (*disconnected*) систем на платформе *.NET*.

Технология *ADO.NET* ориентирована на приложения *N-tier* - архитектуру многоуровневых приложений, которая в настоящее время стала фактическим стандартом для создания распределенных систем.

Основные отличительные особенности *ADO.NET*:

- *ADO* расширяет концепцию объектов-наборов записей в базе данных новым типом *DataSet*, который представляет локальную копию сразу множества взаимосвязанных таблиц. При помощи объекта *DataSet* пользователь может локально производить различные операции с содержимым базы данных, будучи физически рассоединен с СУБД, и после завершения этих операций передавать внесенные изменения в базу данных при помощи соответствующего "адаптера данных" (data adapter);
- в *ADO.NET* реализована полная поддержка представления данных в *XML* -совместимых форматах. В *ADO.NET* сформированные для локальной обработки наборы данных представлены в формате *XML* (в этом же формате они и передаются с сервера баз данных). Данные в форматах *XML* очень удобно передавать при помощи обычного *HTTP*, решает многие проблемы с установлением соединений через брандмауэры;
- *ADO.NET* - это библиотека управляемого кода и взаимодействие с ней производится как с обычной сборкой *.NET*. Типы *ADO.NET*используют возможности управления памятью *CLR* и могут использоваться во многих *.NET* - совместимых языках. При этом обращение к типам *ADO.NET* (и их членам) производится практически одинаково вне зависимости от того, какой язык используется.

Все типы *ADO.NET* предназначены для выполнения единого набора задач:

- установить соединение с хранилищем данных;
- создать и заполнить данными объект *DataSet* ;
- отключиться от хранилища данных и вернуть изменения, внесенные в объект *DataSet* обратно в хранилище данных.

Объект DataSet - это *тип данных*, представляющий локальный набор таблиц и информацию об отношениях между ними.

DataSet - набор связанных таблиц. На практике можно создать на клиенте *объект DataSet*, который будет представлять полную копию удаленной *базы данных*.

После создания объекта *DataSet* и его заполнения данными можно программными средствами производить запросы к нему и перемещаться *по* таблицам, выполнять все *операции*, как при работе с обычными базами данных: добавлять в таблицы новые записи, удалять и изменять существующие, применять к ним фильтры и т.п. После того как клиент завершит внесение изменений,*информация* о них будет отправлена в *хранилище данных* для обработки.

Создание *DataSet* осуществляется при помощи управляемого провайдера (managed provider).

Управляемый провайдер - это набор классов, реализующих интерфейсы, определенные в пространстве имен *System.Data*.

Речь идет об интерфейсах IDbCommand, IDbDataAdapter, IDbConnection и IDataReader (рисунок 4.1).

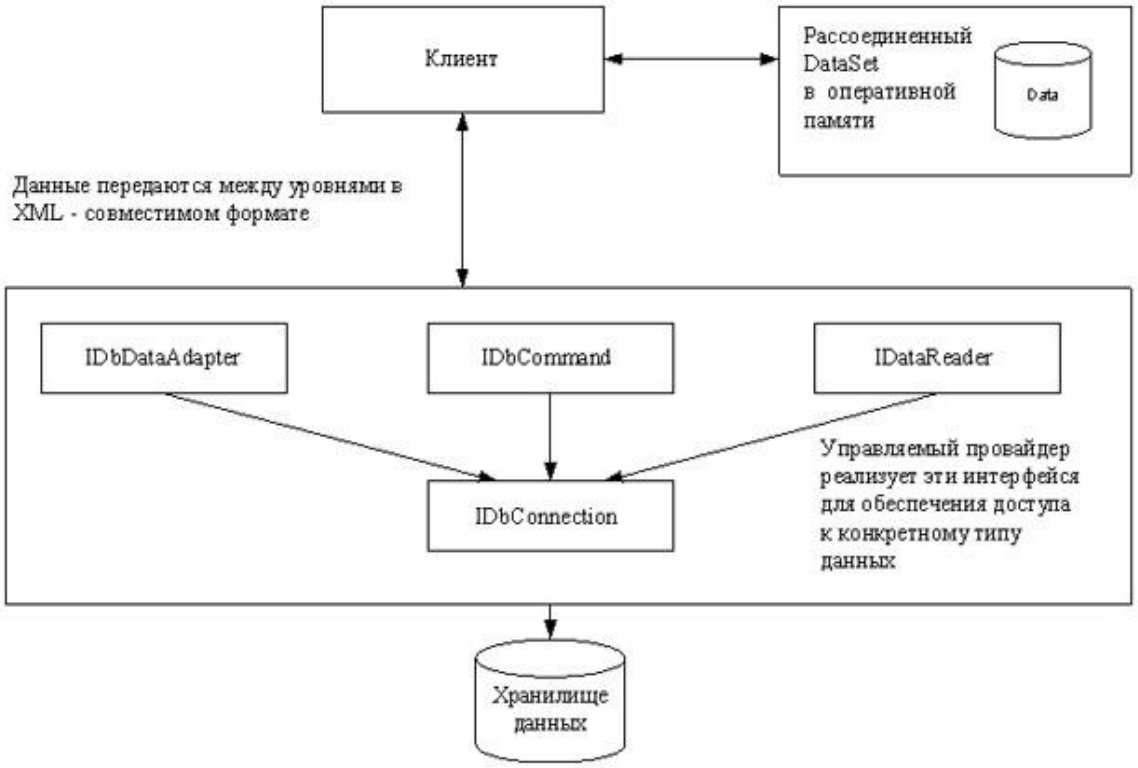

Рис. 4.1. Взаимодействие клиента с управляемыми провайдерами

В состав *ADO.NET* включены два управляемых провайдера: *провайдер SQL* и *провайдер OleDb*. *Провайдер SQL* специально оптимизирован под взаимодействие с *Microsoft SQL Server* версии 7.0 и последующих. Для других источников данных предлагается использовать *провайдер OleDb*, который можно использовать для обращения к любым хранилищам данных, поддерживающим протокол *OLE DB*. Следует отметить, что *провайдер OleDb* работает при помощи "родного" *OLE DB* и требует возможности взаимодействия при помощи *COM*.

Все возможности *ADO.NET* заключены в типах, определенных в соответствующих пространствах имен. Краткий обзор главных пространств имен *ADO.NET* представлен в таблице 4.1.

Таблица 4.1. Пространства имен ADO.NET

Пространство имен	Описание
System.Data	Главное пространство имен ADO.NET. В нем определены типы, представляющие таблицы, столбцы, записи, ограничения и тип - DataSet.
System.Data.Common	Определены типы, общие для всех управляемых провайдеров. Многие из них выступают в качестве базовых классов для классов из пространств имен для провайдеров SQL и OleDb
System.Data.OleDb	В этом пространстве имен определены типы для установления соединений с OLE DB-совместимыми источниками данных, выполнения к ним SQL-запросов и заполнения данными объектов DataSet.
System.Data.SqlCIient	В этом пространстве имен определены типы, которые составляют управляемый провайдер SQL.
System.Data.SqlTypes	Представляют собой "родные" типы данных Microsoft SQL Server.

Все пространства имен *ADO.NET* расположены в одной сборке - System.Data.dll. Это означает, что в любом проекте, использующем *ADO.NET*, мы должны добавить ссылку на эту сборку.

В любом приложении *ADO.NET* необходимо использовать, *по* крайней мере, одно *пространство* имен - *System.Data*. Кроме того, практически во всех ситуациях требуется использовать либо *пространство* имен *System.Data.OleDb* или *System.Data.SqlClient* - для установления соединения с источником данных.

Типы пространства имен *System.Data* предназначены для представления данных, полученных из источника (но не для установления соединения непосредственно с источником).

В основном эти типы представляют собой *объектные представления* примитивов для работы с базами данных - таблицами, строками, столбцами, ограничениями и т. п. Наиболее часто используемые типы *System.Data* представлены в таблице 4.2.

Таблица 4.2. Типы пространства имен System.Data

Тип	Назначение
DataColumnCollection, DataColumn	DataColumn представляет один столбец в объекте DataTable, DataColumnCollection - все столбцы
ConstraintCollection, Constraint	Constraint - объектно-ориентированная оболочка вокруг ограничения (например, внешнего ключа или уникальности), наложенного на один или несколько DataColumn, ConstraintCollection - все ограничения в объекте DataTable
DataRowCollection, Data Row	DataRow представляет единственную строку в DataTable, DataRowCollection - все строки вDataTable
DataRowView, Data View	DataRowView позволяет создавать настроенное представление единственной строки, DataView - созданное программным образом представление объекта DataTable, которое может быть использовано для сортировки, фильтрации, поиска, редактирования и перемещения
DataSet	Объект, создаваемый в оперативной памяти на клиентском компьютере. DataSet состоит из множества объектов DataTable и информации об отношениях между ними

ForeignKeyConstraint, UniqueConstraint	ForeignKeyConstraint представляет ограничение, налагаемое на набор столбцов в таблицах, связанных отношениями первичный - внешний ключ. UniqueConstraint - ограничение, при помощи которого гарантируется, что в столбце не будет повторяющихся записей
DataRelationCollection, DataRelation, DataTableCollection, DataTable	Тип DataRelationCollection представляет набор всех отношений (то есть объектовDataRelation) между таблицами в DataSet. Тип DataTableCollection представляет набор всех таблиц (объектов DataTable) в DataSet

В традиционных системах клиент-*сервер* при запуске приложения пользователем автоматически устанавливается *связь* с базой данных, которая поддерживается в "активном" состоянии до тех пор, пока *приложение* не будет закрыто. Такой метод работы с данными становится непрактичным, поскольку подобные приложения трудно масштабируются. Например, такая прикладная система может работать достаточно быстро и эффективно при наличии 8-10 пользователей, но она может стать полностью неработоспособной, если с ней начнут работать 100, 200 и более пользователей. Каждое открываемое соединение с базой данных "потребляет" достаточно много системных ресурсов сервера, они становятся занятыми поддержкой и обслуживанием открытых соединений, их не остается на процессы непосредственной обработки данных.

При разработке прикладных систем в сети *Интернет* (*Web* -приложения) необходимо добиваться максимальной масштабируемости. Система должна работать одинаково эффективно как с малым, так и с большим числом пользователей.

По этой причине, в *ADO.NET* используется модель работы пользователя в отрыве от источника данных. Приложения подключаются к базе данных только на небольшой промежуток времени. Соединение устанавливается только тогда, когда клиент удаленного компьютера запрашивает на сервере данные. После того, как *сервер* подготовил необходимый набор данных, сформировал и отправил их клиенту в виде *WEB* - страницы, *связь* приложения с сервером сразу же обрывается, и клиент просматривает полученную информацию уже не в связи с сервером. При работе в сети *Интернет* нет необходимости поддерживать постоянную "жизнеспособность" открытых соединений, поскольку неизвестно, будет ли конкретный клиент вообще далее взаимодействовать с источником данных. В таком случае целесообразнее сразу освобождать занимаемые серверные ресурсы, что обеспечит обслуживание большего количества пользователей. Модели доступа к данным представлена на рисунке 4.2.

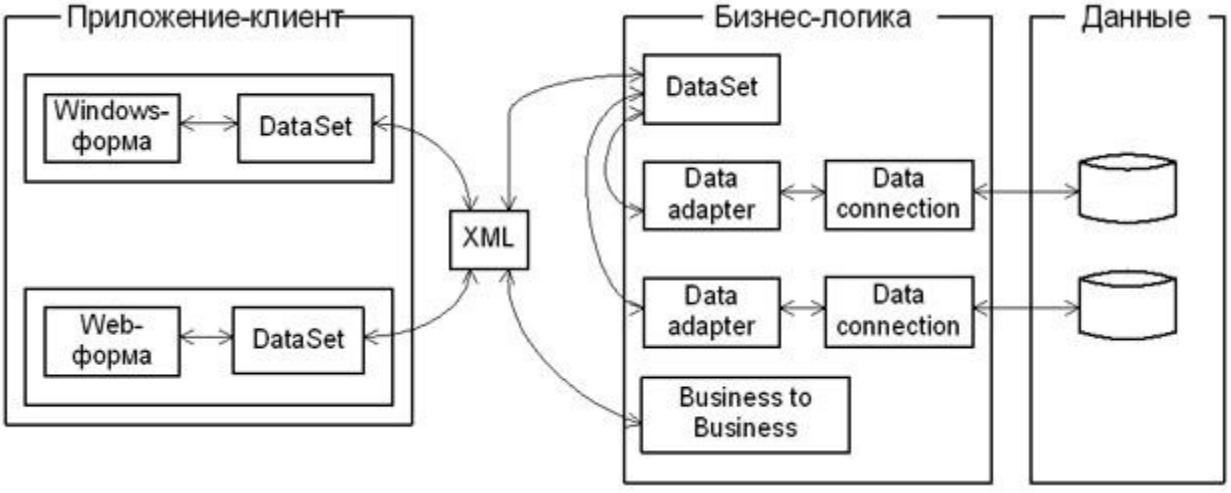

Рис. 4.2. Модель доступа к данным в ADO.NET

В объектной модели *ADO.NET* можно выделить несколько уровней.

Уровень данных. Это *по* сути дела базовый уровень, на котором располагаются сами данные (например, таблицы *базы данных MS SQL Server*). На данном уровне обеспечивается физическое хранение информации на магнитных носителях и манипуляция с данными на уровне исходных таблиц (*выборка*, *сортировка*, добавление, удаление, обновление и т. п.).

Уровень бизнес-логики. Это набор объектов, определяющих, с какой базой данных предстоит установить *связь* и какие действия необходимо будет выполнить с содержащейся в ней информацией. Для установления связи с базами данных используется *объект*DataConnection. Для хранения команд, выполняющих какие либо действия над данными, используется *объект* DataAdapter. И, наконец, если выполнялся процесс выборки информации из *базы данных*, для хранения результатов выборки используется *объект*DataSet. *Объект* DataSet представляет собой набор данных "вырезанных" из таблиц основного хранилища, который может быть передан любой программе-клиенту, способной либо отобразить эту информацию конечному пользователю, либо выполнить какие-либо манипуляции с полученными данными.

Уровень приложения. Это набор объектов, позволяющих хранить и отображать данные на компьютере конечного пользователя. Для хранения информации используется уже знакомый нам *объект* DataSet, а для отображения данных имеется довольно большой набор элементов управления (DataGrid, TextBox, ComboBox, Label и т. д.). В *Visual Studio .Net* можно вести разработку двух типов приложений. В первую *очередь* это традиционные *Windows* -приложения (на основе *Windows* -форм), которые реализованы в виде exe-файлов, запускаемых на компьютере пользователя. Ну и конечно, *Web* -приложения (на основе *Web* -форм), которые работают в оболочке браузера. Как видно из рисунка 8.2, для хранения данных на уровне обоих типов приложений используется*объект* DataSet.

Обмен данными между приложениями и уровнем бизнес-логики происходит с использованием формата *XML*, а средой передачи данных служат либо локальная *сеть* (*Интранет*), либо глобальная *сеть* (*Интернет*).

В *ADO.NET* для манипуляции с данными могут использоваться команды, реализованные в виде *SQL* -запросов или хранимых процедур (DataCommand). Например, если необходимо получить некий набор информации *базы данных*, вы формируете команду SELECTили вызываете хранимую процедуру *по* ее имени.

Когда требуется получить набор строк из *базы данных*, необходимо выполнить следующую последовательность действий:

- открыть соединение (connection) с базой данных;
- вызвать на исполнение метод или команду, указав ей в качестве параметра текст *SQL* -запроса или имя хранимой процедуры;
- закрыть соединение с базой данных.

Связь с базой данных остается активной только на достаточно короткий срок - на период выполнения запроса или хранимой процедуры.

Когда *команда* вызывается на *исполнение*, она возвращает либо данные, либо *код ошибки*. Если в команде содержался *SQL -запрос*на выборку - SELECT, то *команда* может вернуть набор данных. Вы можете выбрать из *базы данных* только определенные строки и колонки,

используя *объект* DataReader, который работает достаточно быстро, поскольку использует курсоры read-only, forward-only.

Если требуется выполнить более чем одну операцию с данными, например, получить некоторый набор данных, а затем скорректировать его, - то необходимо выполнить последовательность команд. Каждая *команда* выполняется отдельно, последовательно одна за другой. Между выполняемыми командами соединение с базой отсутствует. Например, чтобы получить данные из базы - открывается *связь*, выбираются данные, затем *связь* закрывается. Когда выполняется обновление базы после корректировки информации пользователем, снова открывается *связь*, выполняется обновление данных в исходных таблицах и *связь*снова закрывается.

Команды работы с данными могут содержать параметры, т. е. могут использоваться параметризированные запросы, как, например, следующий *запрос*:
SELECT * FROM customers WHERE (customer_id=@customerid)

Значения параметров могут задаваться динамически, во *время выполнения* приложения.

Как правило, в приложениях необходимо извлечь информацию из *базы данных* и выполнить с ней некоторые действия: показать пользователю на экране монитора, сделать нужные расчеты или послать данные в другой *компонент*. Очень часто, в приложении нужно обработать не одну *запись*, а их набор: *список* клиентов, перечень заказов, набор элементов заказа и т. п. Как правило, в приложениях требуется одновременная работа с более чем одной таблицей: клиенты и все их заказы; *автор* и все его книги, заказ и его элементы, т.е. с набором связанных данных. Причем для удобства пользователя данные требуется группировать и сортировать то *по* одному, то *по* другому признаку. При этом нерационально каждый раз возвращаться к исходной базе данных и заново считывать данные. Более практично работать с некой временной "вырезкой" информации, хранящейся в оперативной памяти компьютера.

Эту роль выполняет набор данных - DataSet, который представляет собой своеобразный *кэш* записей, извлеченных из базового источника. DataSet может состоять из одной или более таблиц, он имеет дело с копиями таблиц из *базы данных* источника. Кроме того, в данном объекте могут содержаться связи между таблицами и некоторые ограничения на выбираемые данные. Структура объекта DataSet приведена на рисунке 4.3.

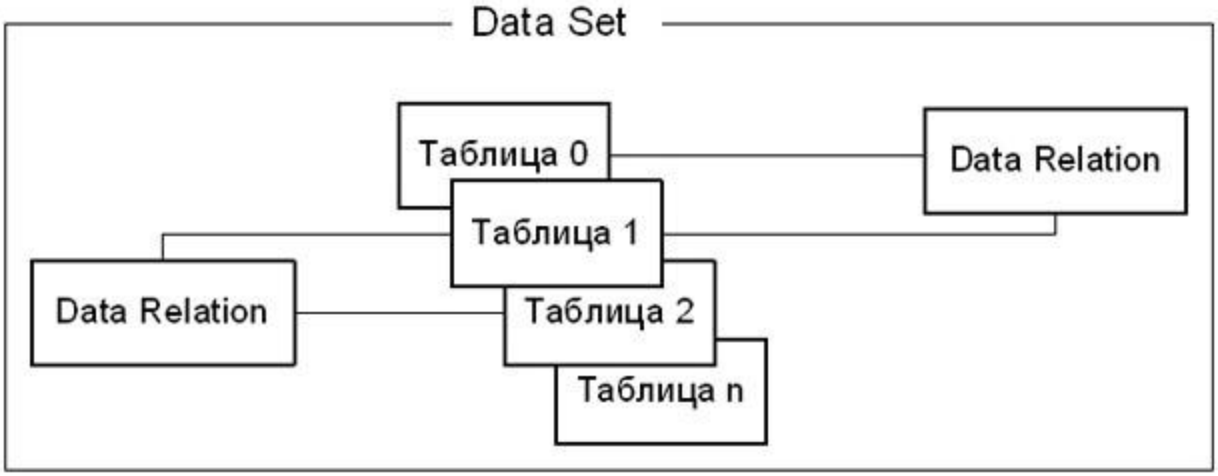

Рис. 4.3. Структура объекта DataSet

Данные в DataSet - это некий уменьшенный вариант основной *базы данных*. Тем не менее, вы можете работать с такой "вырезкой" точно так же, как и с реальной базой. Поскольку каждый *пользователь* манипулирует с полученной порцией информации, оставаясь отсоединенными от основной *базы данных*, последняя может в это время решать другие задачи.

Конечно, практически в любой задаче обработки данных требуется корректировать информацию в базе данных (хотя и не так часто, как извлекать данные из нее). Вы можете выполнить *операции* коррекции непосредственно в DataSet, а потом все внесенные изменения будут переданы в основную базу данных.

Важно отметить то, что DataSet - *пассивный контейнер* для данных, который обеспечивает только их хранение. Что же нужно поместить в этот *контейнер*, определяется в другом объекте - адаптере данных DataAdapter. В адаптере данных содержатся одна или более команд, которые определяют, какую информацию нужно поместить в таблицы объекта DataSet, *по* каким правилам нужно синхронизировать информацию в конкретной таблице DataSet и соответствующей таблицей основной *базы данных* и т. п.*Адаптер* данных обычно содержит четыре команды SELECT, INSERT, UPDATE, DELETE, для выборки, добавления, корректировки и удаления записей.

Например, метод Fill объекта DataAdapter, заполняющего данными *контейнер* DataSet, может использовать в элементеSelectCommand следующий *запрос*:
SELECT au_id, au_lname, au_fname FROM authors

Набор данных DataSet - "независимая" копия фрагмента *базы данных*, расположенная на компьютере пользователя. Причем в этой копии могут быть не отражены те изменения, которые могли внести в основную базу данных другие пользователи. Если требуется увидеть самые последние изменения, сделанные другими пользователями, то необходимо "освежить" DataSet, повторно вызвав метод Fill адаптера данных.

Информация о базе данных

Разрабатываемое *приложение* предназначено для работы с базой данных сотрудников компании. На рисунке 4.4 представлена структура *базы данных*.

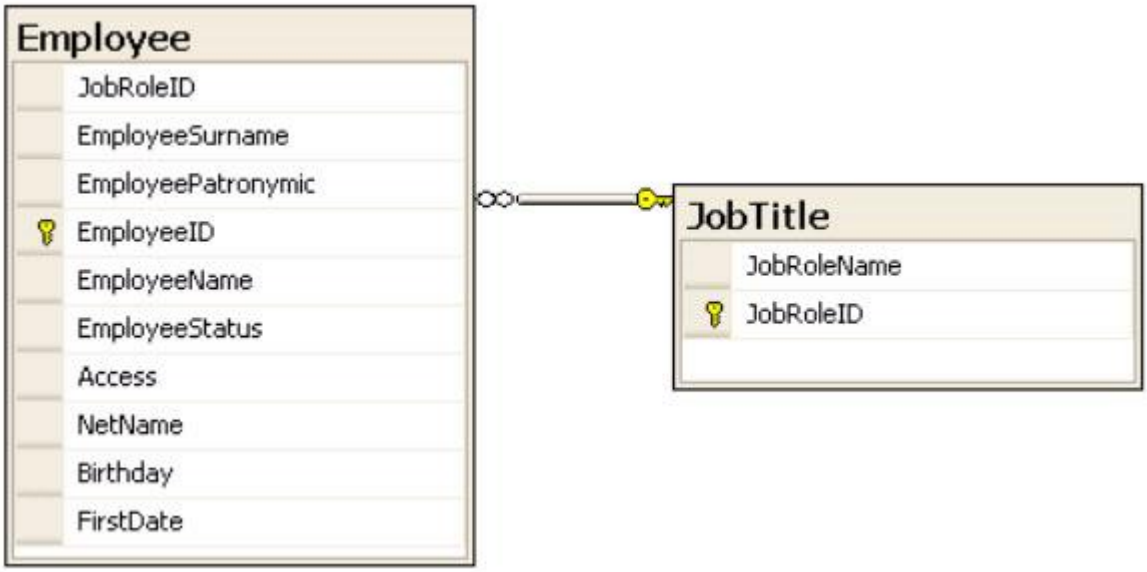

Рис. 4.4. Структура базы данных по сотрудникам компании

База данных включает две таблицы:

- сведения о сотрудниках - *Employee* ;
- справочник должностей - *JobTitle*.

Назначение атрибутов таблицы *Employee* приведены в таблице 4.4

Таблица 4.4. Атрибуты таблицы Employee

Имя атрибута	**Назначение**	**Тип**
EmployeeID	Суррогатный ключ	smallint
JobRoleID	Внешний ключ	smallint
EmployeeSurname	Фамилия	varchar(50)
EmployeeName	Имя	varchar(20)
EmployeePatronymic	Отчество	varchar(20)
EmployeeStatus	Статус	int
Access	Уровень доступа	varchar(20)
NetName	Сетевое имя	varchar(20)
Birthday	Дата рождения	Smalldatetime
FirstDate	Дата приема на работу	smalldatetime

Суррогатный *ключ* EmployeeID, как и все остальные суррогатные ключи *базы данных*, генерируется сервером *базы данных*автоматически, т.е. для него задано свойство IDENTITY для *СУБД MS SQL Server* или AutoNumber для *MS Access*. *Атрибут*JobRoleID является внешним ключом, с помощью которого осуществляется *связь* с таблицей *JobTitle*.

Назначение атрибутов таблицы *JobTitle* приведено в таблице 4.3.

Таблица 4.3. Атрибуты таблицы JobTitle

Имя атрибута	**Назначение**	**Тип**
JobRoleID	Суррогатный ключ	smallint
JobRoleName	Наименование должности	varchar(50)

В рассматриваемом приложении в качестве *СУБД* используется *MS SQL Server 2005*. Создаем соединение проекта с базой данных. Для этого выбираем *пункт меню Tools/Connect to Database*. Появляется окно *AddConnection* (рисунок 4.5)

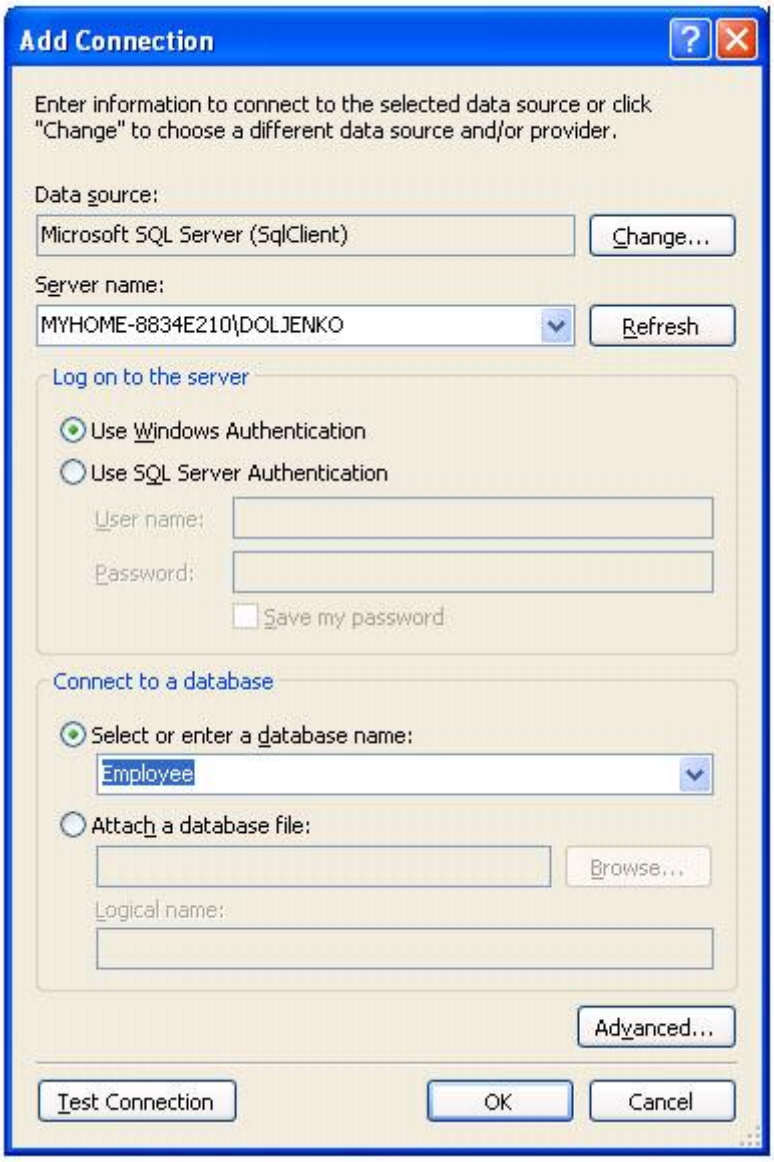

Рис. 4.5. Окно AddConnection

В пункте *"Server name"* задаем имя сервера, которое необходимо узнать у преподавателя (на рисунке 4.5 MYHOME-8834E210\DOLJENKO). В пункте *Select or enter database name* - имя *базы данных*, которое определит преподаватель (на рисунке 4.5 -Employee).

Для проверки правильности подключения к базе данных нажимаем клавишу *"Test Connection"*. При правильном подключении появляется следующее сообщение (рисунок 4.6).

Рис. 4.6. Окно Microsoft Data Link

При нормальном соединении с базой данных можно открыть навигатор *Server Explorer* из *меню View/ Server Explorer* или сочетанием клавиш *ALT+CTRL+S* (рисунок 8.7).

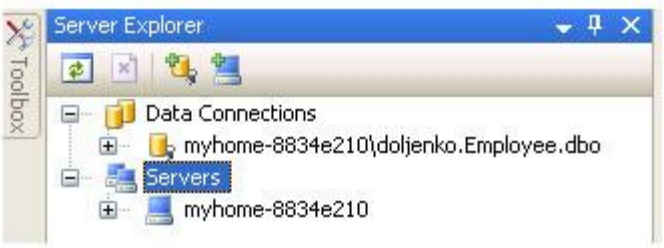

Рис. 8.7. Окно навигатора Server Explorer

Добавим в проект *объект* класса DataSet. Для этого выберем *пункт меню Project/Add New Item. . .* (рисунок 4.8).

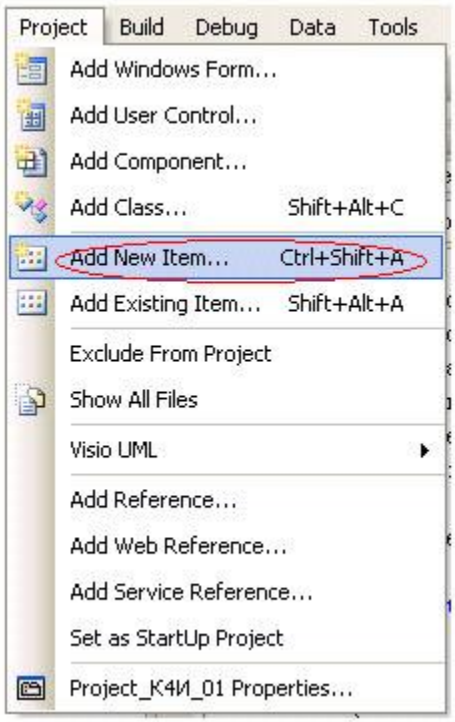

Рис. 4.8. Добавление в проект нового компонента

В окне *Add New Item* (рисунок 4.9) выберем *шаблон DataSet* и присвоим ему имя DataSetEmployee.

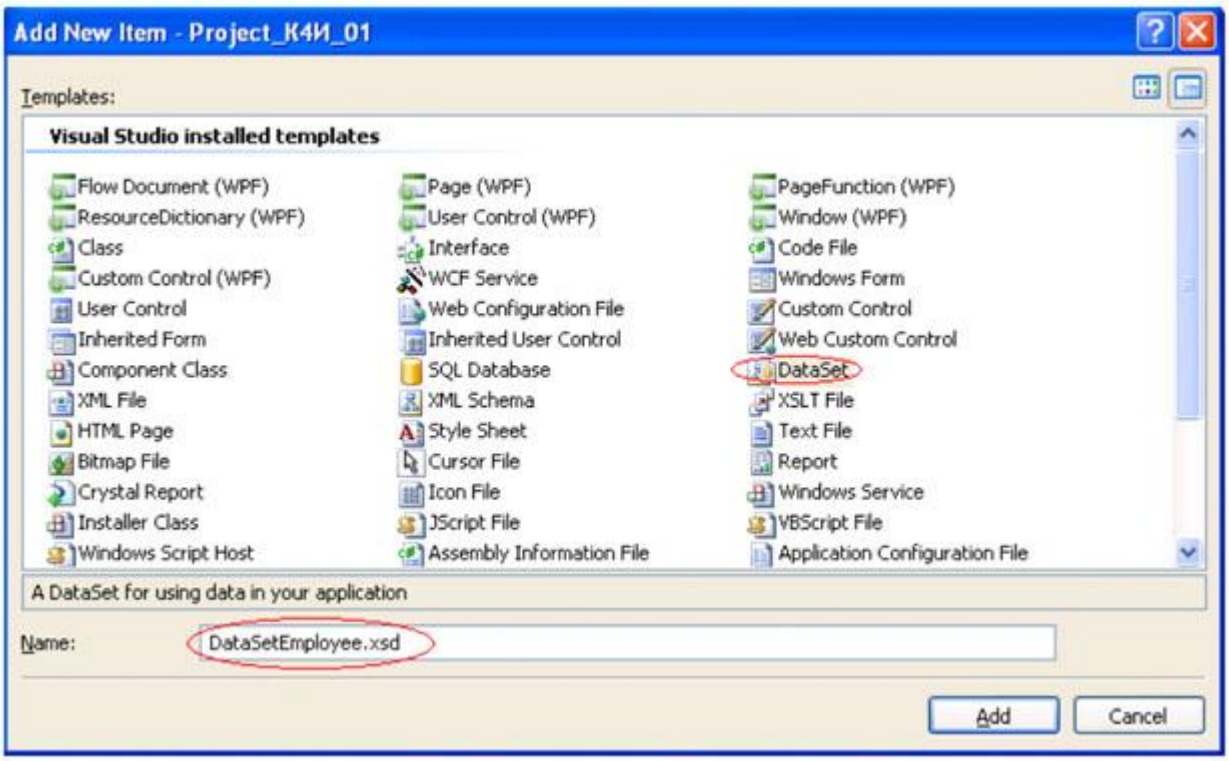

Рис. 4.9. Выбор нового компонента - DataSet

После нажатия кнопки *Add* система генерирует *класс* DataSetEmployee, который добавляется в решение проекта (рисунок 4.10).

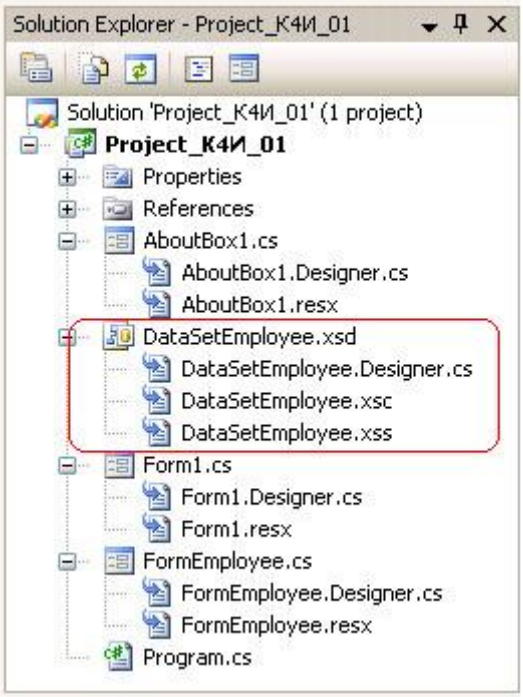

Рис. 4.10. Окно решения проекта с новым компонентом DataSet

Для добавления таблиц *Employee* и *JobTitle* к *DataSet* необходимо перетащить их из окна *Server Explorer* на *поле* графического дизайнера (рисунок 4.11).

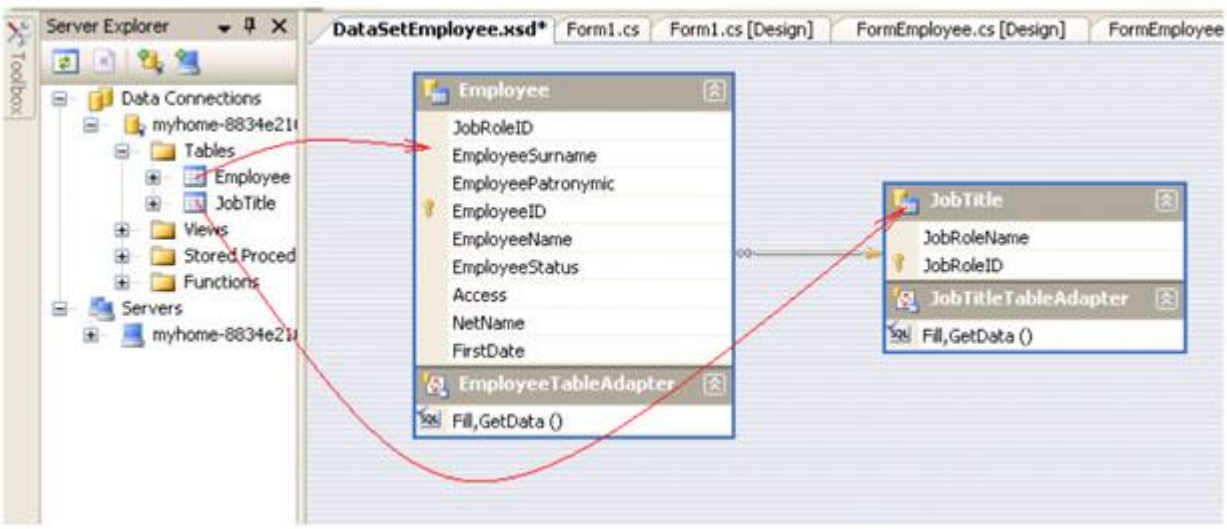

Рис. 4.11. Добавление таблиц к DataSet

В результате будут созданы классы таблиц, адаптеры и методы Fill и GetData.

При формировании класса DataSetEmployee необходимо учесть то, что первичные ключи таблиц *Employee* и *JobTitle* являются суррогатными и автоматически формируются (*ключ* со свойством автоинкремент) источником данных (например, *MS SQL Server*). При формировании новых записей в приложении необходимо обеспечить уникальность первичных ключей для таблиц объектаDataSetEmployee. Это можно обеспечить, задав для ключевых колонок таблиц *Employee* и *JobTitle* следующие свойства:
AutoIncrement = true;
AutoIncrementSeed = -1;
AutoIncrementStep = -1;

Столбец со свойством AutoIncrement равным true генерирует последовательность значений, начинающуюся со значенияAutoIncrementSeed и имеющую шаг AutoIncrementStep. Это позволяет генерировать уникальные значения целочисленного столбца первичного ключа. В этом случае при добавлении новой записи в таблицу будет генерироваться новое *значение*первичного ключа, начиная с -1, -2, -3 и т.д., которое никогда не совпадет с первичным ключом источника данных, т.к. в базе данных генерируются положительные первичные ключи. Свойства AutoIncrementSeed и AutoIncrementStep устанавливаются равными -1, чтобы гарантировать, что когда набор данных будет синхронизироваться с источником данных, эти значения не будут конфликтовать со значениями первичного ключа в источнике данных. При синхронизации DataSet с источником данных, когда добавляют новую строку в таблицу *MS SQL Server 2005* с первичным автоинкрементным ключом, *значение*, которое этот *ключ* имел в таблице *DataSet*, заменяется значением, сгенерированным *СУБД*.

Установка свойств AutoIncrement, AutoIncrementSeed и AutoIncrementStep для колонки первичного ключа EmployeeIDтаблицы *Employee* приведена на рисунке 4.12.

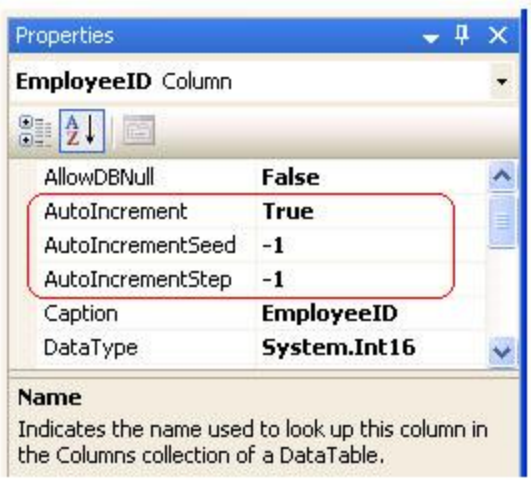

Рис. 4.12. Установка свойств для колонки EmployeeID

Аналогичные установки свойств AutoIncrement, AutoIncrementSeed и AutoIncrementStep необходимо сделать и для колонкиJobTitleID таблицы *JobTitle*.

После создания класса DataSetEmployee и адаптера необходимо создать объекты этих классов, добавив следующий код к файлуFormEmployee.cs.

```
DataSetEmployee dsEmployee = new DataSetEmployee();
DataSetEmployeeTableAdapters.EmployeeTableAdapter daEmployee =
  new Project_К4И_01.DataSetEmployeeTableAdapters.
EmployeeTableAdapter();
DataSetEmployeeTableAdapters.JobTitleTableAdapter daJobTitle =
  new Project_К4И_01.DataSetEmployeeTableAdapters.
JobTitleTableAdapter();
```

После того, как созданы объекты адаптеров данных daEmployee и daJobTitle, а также *объект* класса DataSetEmployee -dsEmployee необходимо создать метод для заполнения объекта dsEmployee из *базы данных* (в рассматриваемом примере *база данных* Employee, созданная в *СУБД MS SQL Server 2005*). Для заполнения данными dsEmployee из *базы данных* Employeeсоздадим метод EmployeeFill( ):

```
public void EmployeeFill()
{   daJobTitle.Fill(dsEmployee.JobTitle);
    daEmployee.Fill(dsEmployee.Employee);
    MessageBox.Show("Метод Fill отработал");}
```

В методе EmployeeFill( ) для объектов класса DataAdapter применяется метод Fill, который производит заполнение таблиц (*JobTitle* и *Employee*) объекта dsEmployee данными из *базы данных*. Метод Fill адаптера данных DataAdapter требует указания в качестве параметров задания соответствующей таблицы *DataSet*, то есть dsEmployee.JobTitle иdsEmployee.Employee.

Метод MessageBox.Show введен в метод EmployeeFill для первоначального тестирования, после которого его нужно убрать.

Вызов метода EmployeeFill необходимо добавить в обработчик события Load для формы FormEmployee, возникающего при нажатии на *пункт меню "Сотрудник"*.

Задание на лабораторную работу

1. Изучите теоретический материал.
2. Создайте класс DataSetEmployee.
3. Для разрабатываемого приложения создайте объекты dsEmployee, daJobTitle и daEmployee.
4. Проведите компиляцию проекта и убедитесь в отсутствии ошибок трансляции.
5. Разработайте метод Fill для заполнение таблиц *DataSet*.
6. Протестировать работу приложения.

Лабораторная работа №5

Модификация, вставка и удаление записей в наборе данных

Цель работы: Изучить основные приемы и способы заполнения объекта DataSet, работы с записями набора данных

Основные концепции обновления наборов данных в Microsoft .NET

Когда в приложении *пользователь* работает с наборами данных DataSet, то фактически он оперирует с записями, оторванными от основной базы и находящимися в памяти компьютера. При такой модели доступа к данным возникает необходимость периодического обновления информации, содержащейся в основной базе данных. Когда *пользователь* вносит какие-либо изменения в записи таблицы объекта DataSet, то DataSet отслеживает и запоминает все эти изменения. *По* мере необходимости все сделанные изменения можно вернуть источнику данных. В течение процесса обновления источника данных происходит несколько событий, которые можно использовать для проверки правильности введенной пользователем информации.

Поскольку набор данных это фактически набор записей, оторванных от основной базы и помещенных в оперативную *память*компьютера, то процесс возврата изменений первоначальному источнику данных является изолированной процедурой и отделен от процесса модификации данных в DataSet.

Модификация источника данных через набор данных происходит в два этапа. На первом этапе модифицируется сам набор данных:*пользователь* добавляет новые записи, изменяет или удаляет существующие. Все эти процессы происходят в копии, оторванной от основного источника данных. На втором шаге необходимо послать эти изменения от оторванного набора данных к первоначальному источнику данных. То есть все изменения в наборе данных не передаются к основному источнику данных автоматически, этот процесс нужного активизировать явно. Это обычно достигается путем вызова метод Update адаптера данных.

На рисунке 5.1 приведен иллюстративный пример обновления полей EmployeeSurname и EmployeeName таблицы Employee в соответствии с двухступенчатой схемой обновления данных в DataSet.

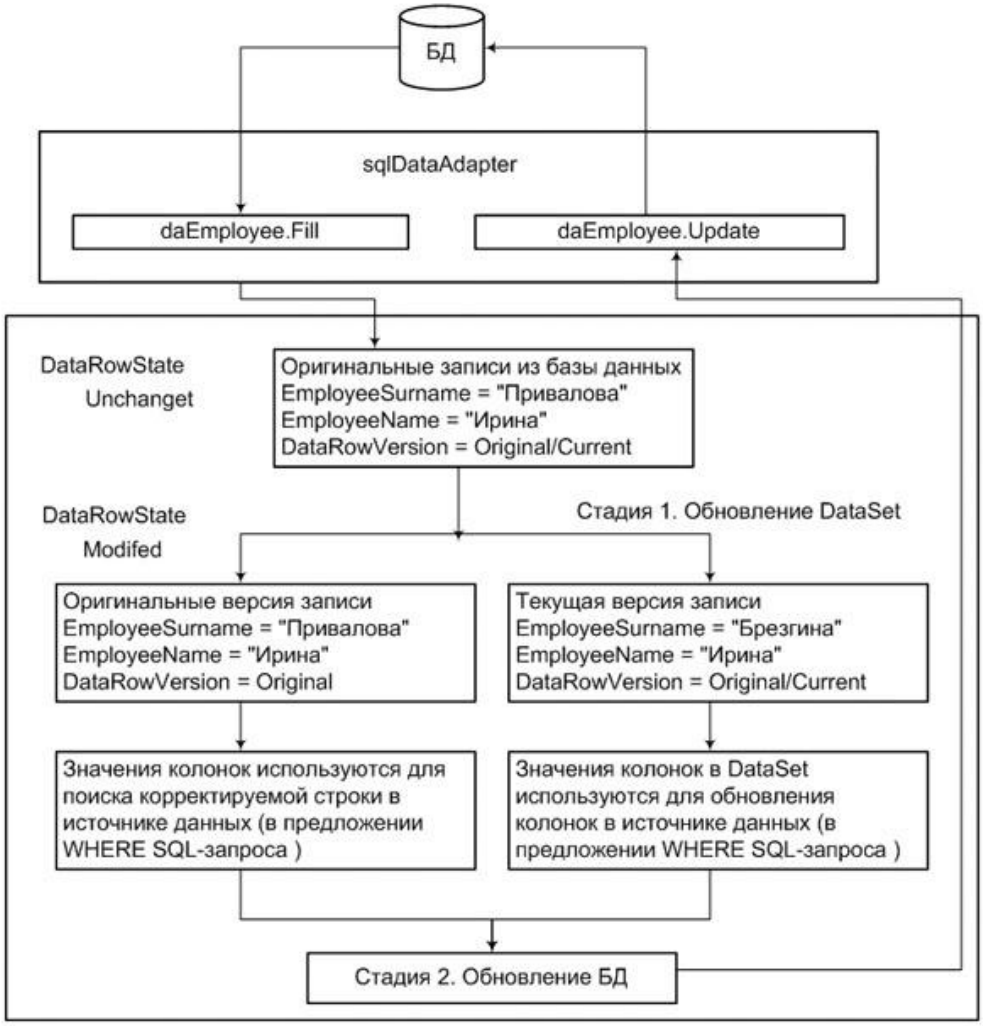

Рис. 5.1. Схема двухступенчатого обновления данных в DataSet

Информация в объекте DataSet доступна через его коллекции. Набор данных содержит коллекцию таблиц (DataTable).*Таблица* содержат коллекции строк (DataRow). Строка содержит коллекцию колонок (DataColumn). Внесение изменений в наборе данных происходит путем навигации *по* элементам коллекции и модификации значений конкретных колонок. В объектеDataSet имеется набор методов, которые позволяют модифицировать только сам набор записей (их используют тогда, когда заведомо известно, что обновления не будут передаваться источнику данных). И есть набор методов, которые позволяют обновить информацию, как в наборе данных, так и в источнике данных.

Например, чтобы удалить записи из таблицы можно вызвать метод RemoveAt коллекции ROWS таблицы данных, которая физически удаляет записи из набора данных. Если набор данных используется только как элемент для отображения информации, и не предполагается возвращать изменения источнику данных или передавать другому приложению, то это наиболее оптимальный*путь*.

Однако если необходимо послать изменения источнику данных или другому приложению, то нужно фиксировать (запоминать) все изменения, которые были сделаны в DataSet. Затем, когда потребуется переслать изменения источнику данных, необходимо будет знать, какие записи были изменены в объекте DataSet и как их найти в источнике данных. Например, если удаляется *запись* в наборе данных, то нужно запомнить ключевую информацию об этой записи, а затем, когда будет вызван метод адаптера данныхDeleteCommand, то ему должны

быть переданы параметры удаленной записи, чтобы он смог определить местонахождение аналогичной записи в источнике данных и удалить именно ее.

Когда выполняется модификация записи, то *пользователь* поочередно изменяет значения отдельных столбцов. Если набор данных содержит ограничения (типа внешних ключей или запрещения присвоения значения null), возможно, что вся *запись* может временно находиться в состоянии ошибки, поскольку *пользователь* не может одновременно присвоить или изменить значения всех столбцов.

Чтобы предотвратить преждевременное нарушение работы программы, можно временно приостановить действие ограничений обновления. Такое действие преследует две цели:

- препятствует преждевременной остановке работы приложения по ошибке, когда пользователь еще не закончил модификацию столбцов записи;
- позволяет обработать некоторые события, в которых можно проверить корректность вводимой пользователем информации.

После того, как *пользователь* завершил обновление всех колонок записи, можно вновь включить действие ограничений.

Когда происходит модификация записи в наборе данных, то могут возникнуть ошибки. Например, столбцу может быть присвоено*значение*, которое имеет несоответствующий *тип данных*, превышает допустимую *размерность*, нарушается *целостность* данных и т. п. Для предотвращения сбоя работы приложения при возникновении ошибочных ситуаций имеется возможность проверки правильности вводимой информации на любой стадии обновления данных.

Информация об изменениях в наборе данных поддержана двумя способами: наличие в каждой строке признака, указывающего на факт внесения изменения в строку (RowState), сохранение нескольких копий (версий) строки (DataRowVersion). На основе этой информации можно однозначно определить, что изменялось в наборе данных и куда адресовать эти изменения в источнике данных.

Объект DataRow имеет свойство DataRowState, в котором хранится *информация* о состоянии строки данных. В таблице 5.1приведены возможные значения свойства DataRowState.

Таблица 5.1. Значения свойства DataRowState объекта DataRow

Значение свойства	**Описание**
Added	Строка была добавлена к коллекции строк DataRowCollection (для этой строки существует только текущая версия в наборе данных и отсутствует соответствующая оригинальная версия). Ее не было в наборе данных на момент последнего вызова метода AcceptChanges
Deleted	Строка была удалена использованием метода DataRow.Delete объекта DataRow
Detached	Строка была создана, но еще не является элементом коллекции DataRowCollection. Объект DataRow находится в этом состоянии сразу после того, как был создан, но еще не добавлен к коллекции, или если он был специально удален из коллекции
Modified	Значение столбца в строке было изменено
Unchanged	Строка не изменялась с тех пор, как был вызван метод AcceptChanges

Наборы данных поддерживают множественные версии записей. *Объект* DataRow имеет свойство DataRowVersion, в котором хранится информации о версии каждой строки. В таблице 5.2 приведены возможные значения свойства DataRowVersion.

Первоначальная (оригинальная) и текущая версии записей используются тогда, когда требуется вернуть изменения источнику данных. Как правило, когда изменения посылаются источнику данных, то все обновления, которые требуется передать базе данных, находятся в текущей версии записи, а *информация* из первоначальной (оригинальной) версии используется для того, чтобы определить местонахождение записи, которую нужно модернизировать в таблице источника данных.

Таблица 5.2. Значения свойства DataRowVersion объекта DataRow

Значение свойства	**Описание**
Current	Текущая версия записи. В этой строке содержатся все изменения, выполненные с момента последнего вызова методаAcceptChanges. Если строка была удалена, то у нее нет текущей версии
Default	Значение по умолчанию (как определено схемой набора данных или источником данных)
Original	Первоначальная (оригинальная) версия записи. Состояние записи до того, как в нее были внесены какие-либо изменения. Практически эта та версия записи, которая была получена от источника данных
Proposed	Промежуточная версия записей, которая существует временно (в режиме внесения в нее изменений). То есть она существует в промежуток времени между вызовами методов начала редактирования (BeginEdit) и завершением редактирования (EndEdit). Вы можете обратиться к этой версии записи в обработчике события RowChanging. Вызов метода CancelEdit полностью отменит все внесенные в строку изменения и удалит промежуточную версию строки данных

Например, если *пользователь* изменил *значение* первичного ключа записи, то в этом случае, если не сохранить оригинальную*запись*, будет просто невозможно определить местонахождение соответствующей записи в источнике данных, чтобы выполнить в ней замену модифицированных пользователем колонок. Перед инициализацией процедуры обновления можно сравнить первоначальную (оригинальную) версию записи с записью в источнике данных, чтобы определить, вносил ли в нее кто-нибудь изменения с тех пор, как она была загружена в набор данных.

Промежуточная версия может быть задействована тогда, когда необходимо выполнить проверку правильности введенной информации перед тем, как изменить текущую версию.

Первоначальная (оригинальная) или текущая версия строки существуют не всегда. Например, когда в таблицу набора данных добавляется новая строка, то первоначальной версии не существует, поскольку этой строки еще нет в источнике данных (существует только текущая версия). Точно так же, если строка была удалена с использованием метода Delete, то остается только первоначальная версия строки, текущая версия отсутствует.

Для проверки существования определенной версии записи можно использовать свойство строки данных HasVersion. Из программы можно обратиться к любой версии записи, передавая *значение* DataRowVersion как дополнительный *параметр* при обращении к значению столбца.

Обычно при работе с приложением модифицируются не все записи в наборе данных. Наборы данных и таблицы данных поддерживают метод GetChanges, что позволяет возвратить только те строки, в которые вносились изменения.

Можно создать любые подмножества измененных записей, используя метод GetChanges для любой таблицы данных (DataTable.GetChanges) или для всего набора данных (Dataset.GetChanges). Если был вызван метод для таблицы данных, то в результате его работы будет возвращена копия таблицы только с измененными записями. Точно так же, если был вызван метод для набора данных, то будет сформирован новый набор данных только с измененными записями этого набора. *Вызов метода*GetChanges без указания статуса строк, позволит получить все измененные записи. Если методу GetChanges передать в качестве параметра статус строки (DataRowState), тогда можно получить нужное *подмножество* измененных записей: только добавленные записи, записи, отмеченные для удаления, отдельные записи (не являющиеся элементами коллекции строк), модифицированные записи.

Получение подмножества измененных записей особенно полезно тогда, когда требуется переслать записи другому компоненту программы для дополнительной обработки. Вместо того чтобы посылать полный набор данных, можно передать только те записи, которые нуждаются в дополнительной обработке, тем самым уменьшить *поток* передаваемой между компонентами информации.

В процессе появления изменений в наборе данных, изменяется и свойство RowState для отдельных записей. В приложении можно обеспечить *доступ*, как к первоначальным, так и к текущим версиям записей, используя свойство RowVersion.

Если все изменения, внесенные пользователем в набор данных, переданы источнику данных, то после этого нет необходимости поддерживать информацию о статусе и версиях записей. Как правило, есть два события, после которых набор данных и *источник данных* находятся в полностью синхронизированном состоянии:

- сразу после того, как информация была загружена из источника данных в набор данных;
- после передачи всех изменений от набора данных до источника данных.

Можно завершить поддержку обработки изменений набора данных, вызвав метод AcceptChanges (принять изменения). Как правило, метод AcceptChanges вызывается в приложении в двух случаях:

- после загрузки набора данных. Если набор данных был загружен путем вызова метода Fill в адаптер данных, то адаптер данных автоматически завершает процедуру поддержки изменений;
- после того, как изменения в наборе данных были пересланы другому процессу, типа *XML Web-service*.

Работа метода AcceptChanges приводит к следующему результату:

- значения колонок текущей (current) версии записей переносятся в колонки первоначальной (original) версии записей (поверх имеющихся там данных);
- удаляются все строки, свойство которых RowState имеет значение Deleted ;
- свойству всех записей RowState присваивается значение *Unchanged*.

Метод AcceptChanges доступен на трех уровнях. Можно вызвать данный метод для строки (для объекта DataRow), но тогда завершаются изменения только в одной строке. Можно также вызвать этот метод для таблицы (для объекта DataTable), в результате чего изменения

будут зафиксированы во всех строках таблицы. И, наконец, метод можно применить ко всему набору данных (для объекта DataSet), вследствие чего будут завершены все изменения во всех записях всех таблиц набора данных.

В таблице 5.3 показано, какие изменения будут завершены в зависимости от того, к какому объекту адресован методAcceptChanges.

Таблица 5.3. Результат работы метода AcceptChanges

Метод	**Результат работы метода**
DataRow.AcceptChanges	Завершение изменений только в определенной строке
DataTable.AcceptChanges	Завершение изменений во всех строках определенной таблицы
DataSet.AcceptChanges	Завершение изменений во всех строках, во всех таблицах набора данных

С методом AcceptChanges связан другой метод - RejectChanges, который выполняет противоположное действие - отменяет все изменения. При вызове метода RejectChanges происходит *копирование* столбцов строки первоначальной версии в строку текущей версии, а свойству RowState каждой записи присваивается *значение Unchanged*.

Модификация, вставка и удаление записей в наборе данных

После того, как *объект* DataSet заполнен данными, пользователи приложения обычно модифицируют полученную информацию, и затем эти изменения пересылаются источнику данных. Так как отдельные записи в DataSet представлены объектом DataRow, то все изменения в набор данных производятся путем модификации, добавления и удаления индивидуальных строк.

Чтобы выполнить редактирование какого-либо значения в наборе данных, необходимо обратиться к конкретному столбцу в определенной строке. К определенному значению в наборе данных можно обратиться следующим образом:

- через индексы таблиц, строк и коллекций столбцов;
- передавая в соответствующие коллекции наименования таблиц и столбцов как строковые переменные.

Модифицирование записей в типизированных и нетипизированных наборах данных осуществляется различными способами.

Наименования таблиц и столбцов в наборах данных без контроля типов (нетипизированные наборы) не доступны во времени разработки приложения и к ним нужно обращаться через их соответствующие индексы.

В типизированных наборах данных используется раннее *связывание*, за счет чего имена таблиц и столбцов доступны на этапе разработки приложения. Это позволяет создавать более удобочитаемый программный код. В следующем примере показано, как можно модифицировать данные в столбцах JobRoleID и EmployeeSurname пятой записи таблицы Employee в наборе данныхdsEmployee1.

```
dsEmployee1.Employee[4].JobRoleID = 2;
dsEmployee1.Employee[4].EmployeeSurname = "Иванова";
```

Для добавление записей в набор данных необходимо создать новую строку и добавить ее к коллекции DataRow соответствующей таблицы. Следующий пример показывает, как вставить дополнительные строки в *объект* DataTable - Employee набора данныхdsEmployee.

Добавление записи в набор данных происходит следующим образом.

1. Вызовите метод таблицы данных NewRow, который создает новую пустую запись. Эта новая запись наследует коллекцию столбцов DataColumnCollection от исходной таблицы данных Employee.
   DataRow employeeRow = Employee.NewRow( );
2. Присвойте колонкам этой строки требуемые значения .
3. employeeRow[JobRoleID] = 1;
4. employeeRow[EmployeeSurname] = "Ларина";
5. employeeRow[EmployeeName] = "Татьяна";
   employeeRow[EmployeePatronymic] = "Ивановна";
6. Добавьте новую запись в таблицу, вызвав метод Add объекта
   dsEmployee.Employee.Rows.Add(employeeRow);

Удаление записей из набора данных. Если в приложении после удаления записи в наборе данных требуется сделать то же самое и в источнике данных, то необходимо использовать метод Delete таблицы данных. Если ваше *приложение* не должно пересылать обновления назад источнику данных, тогда возможно прямое удаление записи из коллекции строк.

Для удаления записи из таблицы данных вызовите метод Delete. Этот метод физически не удаляет *запись* из набора данных; он просто меняет ее статус - отмечает как удаленную *запись*, и она становится невидимой (перестает быть доступной конечному пользователю приложения).

Следующий пример показывает, как вызвать метод Delete для удаления первой строки таблицы Employee набора данныхdsEmployee:
dsEmployee.Employee.Rows[0].Delete( );

Когда происходит обновление (модификация) записей, то в объекте DataTable активизируются определенные события, в обработчике которых можно запрограммировать требуемые действия как до, так и после обновления записи.

Завершение изменений в наборе данных. В наборе данных при внесении изменений в записи (обновление, вставка, удаление) поддерживается и первоначальная (оригинальная), и текущая версии записей. Кроме того, в свойстве RowState каждой записи установлен признак (*указатель*), находятся ли *запись* в первоначальном состоянии или она была изменена. Эта *информация*необходима, когда требуется найти определенную версию строки. Как правило, в приложениях возникает необходимость передачи между компонентами только измененных записей. После того, как проведена нужная обработка всех измененных записей, можно завершить процедуру обновления, вызвав метод набора данных AcceptChanges.

Следующий пример показывает, как вызвать метод AcceptChanges для завершения изменений в таблице Employee после передачи обновлений источнику данных.
daEmployee.Update(dsEmployee, "Employee");
dsEmployee.Employee.AcceptChanges( );

Обновление баз данных из наборов данных. После того, как были сделаны изменения в наборе данных, их можно передать источнику данных. Обычно это делается путем вызова метода Update адаптера данных. Данный метод в цикле просматривает все записи таблицы и определяет, какие *операции* необходимо выполнить (обновление, добавление или удаление), и, если таковые вообще имеются, выполняет соответствующие команды. Если при просмотре очередной строки метода Update адаптера данных обнаруживает, что строка изменялась, то

он автоматически вызовет надлежащую команду данных (DataCommand) и обновит на ее основе соответствующую строку в базе данных.

Структура *SQL* -запроса команды данных будет следующая:

- будет задействован *SQL -запрос с предложением* UPDATE. Адаптер данных знает, что предложение UPDATE применяется для тех строк, свойства RowState которых имеет значение Modified ;
- в *SQL* -запрос будет включено предложение WHERE, указывающее, что обновляться будет строка, для которой, напримерEmployeeID = 4. Эта часть *SQL* -запроса отличает целевую строку от всех других строк, потому что поле EmployeeIDявляется первичным ключом таблицы базы данных. Информация для предложения WHERE будет получена не из текущей, а из оригинальной (первоначальной) версии записи (DataRowVersion.Original), потому что в текущей версии записи пользователь мог изменить значение первичного ключа.
- в *SQL* -запрос будет включено предложение SET, для того, чтобы установить новые значения столбцов в соответствующей записи таблицы базы данных.

Обработка исключительных ситуаций при обновлении источника данных из набора данных. После того, как набор данных был изменен и эти изменения приняты, *приложение* должно вернуть новые данные назад источнику данных. Чтобы модернизировать*источник данных* содержанием набора данных вызывается метод Update адаптера данных. *Адаптер* данных выполнит соответствующую команду (insert, update или delete), используя *значение* свойства RowState каждой строки набора данных.

Процедура модернизации источника данных зависит от требований конкретного приложения, но в любом случае должны быть выполнены следующие шаги.

1. Вызов метода Update адаптера данных должен выполняться внутри блока Try...Catch.
2. Если возникла исключительная ситуация, то нужно определить при обработке какой строки набора данных произошла ошибка.
3. Исправление возникшей проблемы в строке данных (либо программным способом, либо, представив пользователю ошибочную строку для внесения в нее изменений), и затем - повторная попытка модернизации источника данных.

Следующий пример показывает, как делать модернизацию источника данных внутри блока Try...catch содержанием набора данных с именем.

```
try
{ daEmployee.Update(dsEmployee, "Employee"); }
catch (Exception e)
{
  // Код обработки ошибочной ситуации.
  String err = e.msg;
  MessageBox.Show("Ошибка обновления таблицы базы данных
Employee"  + err);
}
```

Обновление связанных таблиц. Если в наборе данных содержится несколько таблиц, то их необходимо модифицировать индивидуально, вызывая отдельно метод Update каждого адаптера данных. Если таблицы имеют реляционные отношения, то потребуется специфическое обновление *базы данных*. Обычно в этом случае изменяются и родительская *запись* и связанные с ней дочерние записи набора данных. Например, при добавлении новой должности формируется одна или более связанных записей сотрудников.

Если в самой базе данных заданы правила обеспечения целостности, то при обновлении возможно возникновение ошибок. Например, если переслать в базу данных вновь появившиеся дочерние записи раньше, чем была переслана родительская*запись*. Поэтому необходимо выполнять определенную последовательность в пересылке: сначала должны послаться новые родительские записи, а потом дочерние.

Наоборот, если вы удаляете связанные записи в наборе данных, необходимо посылать обновления в базу данных в обратном порядке: сначала от дочерней таблицы, а потом - от родительской. В противном случае при обновлении информации в базе данных возникнет ошибка, потому что правила обеспечения целостности не позволят удалить родительскую *запись*, пока существуют связанные дочерние записи.

Последовательность обновления связанных таблиц. При обновлении связанных таблиц набора данных очень важно соблюдать определенную последовательность, чтобы не нарушить *целостность* данных основного источника и не сгенерировать ошибочную ситуацию. Чтобы предотвратить возможные ошибки нарушения целостности данных необходимо модернизировать *источник данных*в следующей последовательности:

- дочерняя таблица: удалить записи;
- родительская таблица: добавить, изменить или удалить записи;
- дочерняя таблица: добавить или изменить записи.

Для модернизации источника данных из набора данных, который содержит две или более связанных таблиц вызовите методUpdate каждого адаптера данных.

Следующий пример показывает, как модернизировать *источник данных* из набора данных, который содержит связанные таблицы. Чтобы следовать за вышеупомянутой последовательностью, создаются три временных набора данных дочерней таблицы, которые содержат только добавленные, только удаленные и только измененные записи. Метод Update вызывается для каждого поднабора данных внутри блока Try...catch. Если при обновлении возникнут ошибки, то они будут перехвачены и обработаны также внутри этого блока (в обработчике исключительных ситуаций). После обновления источника данных вызывается методAcceptChanges, который "принимает" сделанные изменения и, наконец, удаляются временные наборы данных, чтобы освободить ресурсы.

```
void UpdateAttemp( )
{
//Набор dsl содержит только удаленные записи дочерней таблицы
DataTable dsl = anyDataset.ChildTableName.GetChanges(DataRowState.Deleted);
//Набор ds2 содержит только добавленные записи дочерней таблицы
DataTable ds2 =
anyDataset.ChildTableName.GetChanges(DataRowState.Added);
//Набор ds3 содержит только измененные записи дочерней таблицы
DataTable ds3 =anyDataset.ChildTableName.GetChanges(DataRowState.Modified);
try
{
//Удаление записей дочерней таблицы
DataAdapter2.Update(dsl);
// Добавление и удаление записей родительской таблицы
DataAdapterl.Update(anyDataset, "ParentTable");
// Добавление записей в дочернюю таблицу
DataAdapter2.Update(ds2);
// Обновление записей дочерней таблицы
```

```
DataAdapter2.Update(ds3);
// Завершение процесса модернизации набора данных
anyDataset.AcceptChanges( );
// Удаление временных наборов данных
dsl.Dispose ( );
ds2.Dispose( );
ds3.Dispose( );
}
catch (Exception x)
{
// Обработчик ошибочных (исключительных) ситуаций.
}
}
```

Листинг а.

Модификация данных

Модификация данных в приложении реализуется методом Edit. Для модификации данных необходимо для элементов контроля задать режим редактирования, вызвав метод *DisplayEdit* и запретить *доступ* к списку сотрудников в элементе listBoxEmployee. Активизация режима модификации данных *по* сотруднику осуществляется из пункта *меню* Действие/Изменить.

```
private void Edit ( )
{
        DisplayForm (false);
        this.listBoxEmployee.Enabled = false;
}
```

При проектировании приложения использовалась технология привязки источника (DataSet) данных к элементам управления (TextBox, ListBox, ComboBox). При выводе данных на экранную форму проведенная привязка обеспечивала *корректность* работы приложения. При модификации данных в элементах контроля для текстового поля TextBox механизм связывания автоматически поддерживает синхронизацию между элементом контроля и источником данных. Для списка ComboBox необходимо использовать событие SelectionChangeCommitted, которое им генерируется, когда *пользователь* изменяет выбранный элемент и подтверждает его изменение. Обработчик этого события принимает *аргумент* EventArgs. Этот обработчик целесообразно использовать для получения нового значения из ComboBox, когда *пользователь* его изменит. Для элементов управление обработчики событияSelectionChangeCommitted представлены ниже:

```
private void comboBoxJobRole_SelectionChangeCommitted(object sender, System.EventArgs e)
{
        // Определяем позицию в таблице Employee
        int pos = -1;
        pos = this.BindingContext[dsEmployee1,
"Employee"].Position;
// Изменение в таблице Employee поля JobRoleID при изменении
// выбора должности (comboBoxJobRole)
        dsEmployee1.Employee[pos].JobRoleID =
(short)((DataRowView)comboBoxJobRole.Items[comboBoxJobRole.SelectedIndex])
["JobRoleID"];
}
private void comboBoxAccess_SelectionChangeCommitted(object sender, System.EventArgs e)
{
```

```
            int pos = -1;
            pos = this.BindingContext[dsEmployee1,
"Employee"].Position;
            dsEmployee1.Employee[pos].Access =
comboBoxAccess.SelectedItem.ToString();
}

private void comboBoxStatus_SelectionChangeCommitted(object sender, System.EventArgs e)
{
            int pos = -1;
            pos = this.BindingContext[dsEmployee1,
"Employee"].Position;
            dsEmployee1.Employee[pos].EmployeeStatus =
comboBoxStatus.SelectedIndex;
}
```
Листинг b.

Далее можно изменять данные *по* сотруднику на форме. После изменения данных необходимо сохранить сделанные изменения.

Сохранение данных

Сохранение данных в приложении реализуется методом Save, который активизируется при выборе пункта *меню Действие/Сохранить*. Для сохранения модифицированной информации в приложении необходимо вначале завершить текущее обновление всех связанных с помощью объектов Binding элементов управления *Windows Forms* и источника данных. Для этого необходимо вызвать метод EndCurrentEdit класса BindingManagerBase:

```
bmEmployee.EndCurrentEdit( )
```

Далее формируется *таблица* ds1, в которую включаются только модифицированные строки:

```
DataSetEmployee.EmployeeDataTable ds1 =
(DataSetEmployee.EmployeeDataTable)dsEmployee.Employee.
GetChanges(DataRowState.Modified);
```

Если *таблица* ds1 не пуста (условие - ds1 != null) осуществляется попытка обновления *базы данных* с помощью методаUpdate адаптера daEmployee и далее уничтожается *таблица* ds1. Если обновить базу данных не удается, то формируется*сообщение об ошибке*:

```
if(ds1 != null)
  try
  {
    this.daEmployee.Update(ds1);
    ds1.Dispose( );
    dsEmployee.Employee.AcceptChanges();
  }
  catch(Exception x)
  {
    string mes = x.Message;
   MessageBox.Show("Ошибка обновления базы данных Employee "+mes,
"Предупреждение");
    this.dsEmployee.Employee.RejectChanges();
  }
```

При завершении алгоритма обновления форма *FormEmployee* переводится в режим "только для чтения" с помощью методаDisplayReadOnly и фокус должен быть на элементе управления textBoxSurname.

Полный текст модифицированного метода Save приведен далее:

```
private void Save()
{

   string mes = "";
// Завершение текущих обновлений всех  связанных с помощью
// объектов Binding элементов управления
   bmEmployee.EndCurrentEdit();

///Формирование таблицы, в которую включаются только
// модифицированные строки
   DataSetEmployee.EmployeeDataTable ds1 =
(DataSetEmployee.EmployeeDataTable)dsEmployee.Employee.
GetChanges(DataRowState.Modified);

   if(ds1 != null)
      try
      {
         this.daEmployee.Update(ds1);
         ds1.Dispose( );
         dsEmployee.Employee.AcceptChanges();
      }
      catch(Exception x)
      {
         string mes = x.Message;
         MessageBox.Show("Ошибка обновления базы данных Employee "+mes,
"Предупреждение");
         this.dsEmployee.Employee.RejectChanges();
      }
   DisplayForm(true);
   listBoxEmployee.Enabled = true;
   textBoxSurname.Focus();
}
```

Листинг с.

Протестируйте добавленный в программу код.

Добавление данных

Создание новой записи с данными *по* сотруднику в приложении реализуется методом New. При добавлении новых записей в таблицу Employee *базы данных* необходимо:

1. Создать новую строку
   ```
   DataRow rowEmployee = this.dsEmployee.Employee.NewEmployeeRow();
   ```
2. Сформировать начальные значения для элементов строки
3. rowEmployee["JobRoleID"] = 1;
4. rowEmployee["EmployeeStatus"] = 0;
5. rowEmployee["EmployeeSurname"] = "";
6. rowEmployee["EmployeeName"] = "";

7. rowEmployee["EmployeePatronymic"] = "";
   rowEmployee["Access"] = "не задано";
8. Добавить сформированную строку к таблице Employee
   dsEmployee.Employee.Rows.Add(rowEmployee);
9. Установить активную позицию в таблице Employee на добавленную строку
10. int pos = this.dsEmployee.Employee.Rows.Count - 1;
    this.BindingContext[dsEmployee, "EmployeeID"].Position = pos;
11. Задать режим редактирования формы
    DisplayForm(false);
12. Сделать список сотрудников недоступным для выбора
    listBoxEmployee.Enabled = false;
13. Установить фокус на элементе textBoxSurname
    textBoxSurname.Focus();

После задания данных *по* новому сотруднику необходимо их запомнить в базе данных. Для этого требуется модифицировать методSave, добавив в него фрагмент кода для вставки новых записей перед кодом, модифицирующим записи.

```
/// Формирование таблицы, в которую включаются только добавленные строки
DataSetEmployee.EmployeeDataTable ds2 = (DataSetEmployee.EmployeeDataTable)dsEmployee.Employee.
GetChanges(DataRowState.Added);
if(ds2 != null)
  try
  {
   daEmployee.Update(ds2);
   ds2.Dispose();
    dsEmployee.Employee.AcceptChanges( );
  }
  catch(Exception x)
   {
    string mes = x.Message;
   MessageBox.Show("Ошибка вставки записи в базу данных Employee "+mes, "Предупреждение");
   this.dsEmployee.Employee.RejectChanges();
  }
```

В добавляемом фрагменте проверяется наличие в таблице Employee добавленных строк
ds2 != null

и если они имеются, то производится попытка модифицировать базу данных
daEmployee.Update(ds2);

При вставке новых записей в базу данных метод Update адаптера daEmployee вызывает *SQL* команду INSERT, которая была сформирована при генерации адаптера. Если *первичный ключ* таблицы *базы данных* является суррогатным и генерируется сервером автоматически (задано свойство IDENTITY), которое корректируется в DataSet.

Протестируйте добавленный в программу код.

Удаление данных

Удаление записи с данными *по* сотруднику в приложении реализуется методом Remove. В процессе удаления записей *по*сотрудника используется модальное *диалоговое окно* для вывода сообщения (см. "Создание пользовательских диалоговых окон"), вид которого представлен на рисунке 5.2, для предупреждения пользователя при проведении *операции*.

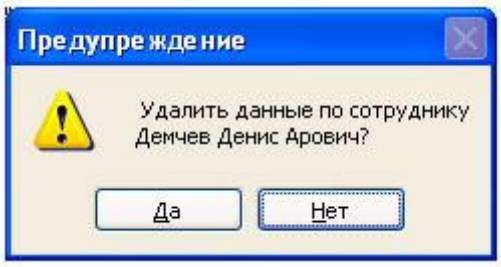

Рис. 5.2. Диалоговое окно предупреждения пользователя.

При выборе пункта *меню Действие/Удалить* формируется событие, которое обрабатывается методомtoolStripButtonRemove_Click. Обработчик вызывает метод Remove, который выполняет удаление данных *по* сотруднику. В методе Remove определяется позиция, которую необходимо удалить в таблице Employee

```
int pos = -1;
pos = this.BindingContext[dsEmployee, "Employee"].Position;
```

Затем формируется строка с фамилией, именем и отчеством, удаляемого сотрудника

```
string mes = textBoxSurname.Text.ToString().Trim() + " "
  + textBoxName.Text.ToString().Trim() + " " +
  textBoxPatronymic.Text.ToString().Trim();
```

и выводится сообщение в диалоговом окне

```
MessageBox.Show(" Удалить данные  \n по сотруднику \n"+ mes+ "?", "Предупреждение",
  MessageBoxButtons.YesNo, MessageBoxIcon.Warning, MessageBoxDefaultButton.Button2);
```

Далее в зависимости от того, какую кнопку нажал *пользователь*, осуществляется удаление записи. Если *пользователь* нажал кнопку*"Да"*, то необходимо выполнить следующий код:

```
this.dsEmployee.Employee.Rows[pos].Delete();
if (this.dsEmployee.Employee.GetChanges(DataRowState.Deleted) != null)
{
  try
  {
    this.daEmployee.Update(dsEmployee.Employee);
    this.dsEmployee.Employee.AcceptChanges();
  }
  catch (Exception x)
  {
    string er = x.Message.ToString();
    MessageBox.Show("Ошибка удаления записи в базе данных Employee " + er,
"Предупреждение");
    this.dsEmployee.Employee.RejectChanges();
  }
}
```

При нажатии кнопки *"Нет"*, то необходимо выполнить код:

```
this.dsEmployee.Employee.RejectChanges();
```

Протестируйте добавленный в программу код.

Отмена изменений

Отмена модификации записи с данными *по* сотруднику в приложении реализуется методом Undo. При отмене изменений необходимо завершить текущие обновления связанных элементов управления
bmEmployee.EndCurrentEdit();

и с помощью метода RejectChanges отменить изменения в источнике данных
dsEmployee.Employee.RejectChanges();

Далее необходимо задать режим *"только для чтения"*, выбор строки в списке сотрудников listBoxEmployee и установить фокус на форме.

Задание на лабораторную работу

1. Изучите теоретический материал.
2. Разработайте методы для модификации, формирования, удаления и сохранения данных по сотрудникам.
3. Разработайте метод для отмены модификации данных по сотрудникам.
4. Протестируйте приложение.

Лабораторная работа №6

Упорядочивание списков и вычисляемые столбцы DataSet

Цель работы: Изучить основные приемы и способы сортировки данных при заполнении данными DataSet, динамическое изменение структуры DataSet, создание вычисляемых столбцов в DataSet и *связывание* их с элементами представления

Основные положения

В результате выполнения лабораторных *работ* 5-6 разработано *приложение* для учета сотрудников предприятия, основная *экранная форма* которого приведена на рисунке 6.1.

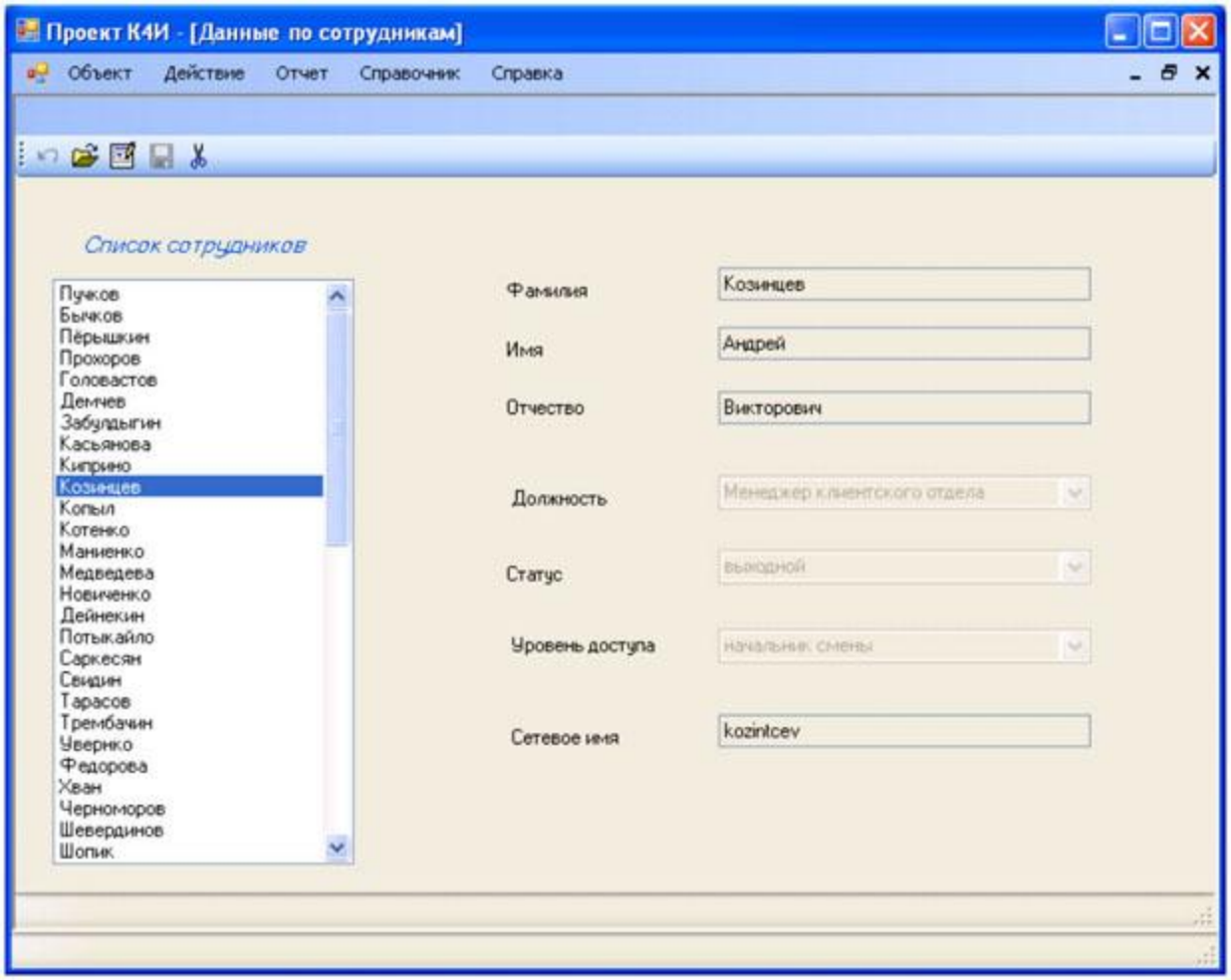

Рис. 6.1. Экранная форма учета сотрудников предприятия

При аттестации разработанного приложения конечными пользователями были сделаны следующие замечания:

- список сотрудников необходимо упорядочить по алфавиту;
- в списке сотрудников необходимо указывать фамилию, имя и отчество.

Упорядочение списка сотрудников по алфавиту

Данные о сотрудниках считываются из таблицы Employee *базы данных* с помощью *SQL* - запроса в *объект* класса DataSet, с которым связан элемент визуализации данных listBoxEmployee. Данные из таблицы Employee *базы данных* заполняют таблицуDataSetEmployee в той последовательности, как они записаны в базе данных и соответственно в той же последовательности и отображаются в элементе визуализации

данных listBoxEmployee. Для упорядочения списка сотрудников по алфавиту достаточно упорядочить данные получаемые из *базы данных* с помощью *SQL* -запроса, т.е. необходимо модифицировать команду Select.

Для формирования новой команды Select необходимо переконфигурировать Data Adapter - daEmployee.

Конфигурирование Data Adapter

1. В дизайнере класса DataSetEmployee выделите EmployeeTableAdapter (рисунок 11.2) и через контекстное меню откройте помощник конфигуратора адаптера *Table Adapter Configuration Wizard* (рисунок 6.3).

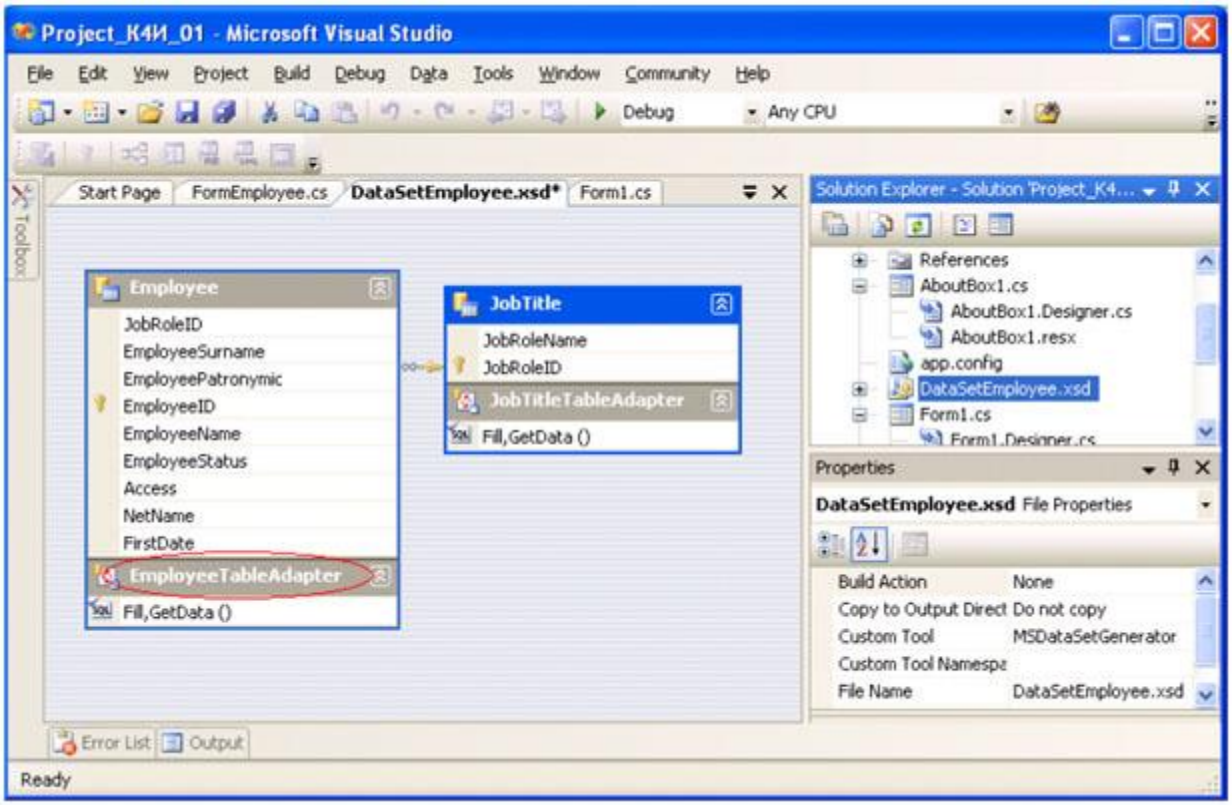

Рис. 6.2. Дизайнер класса DataSetEmployee

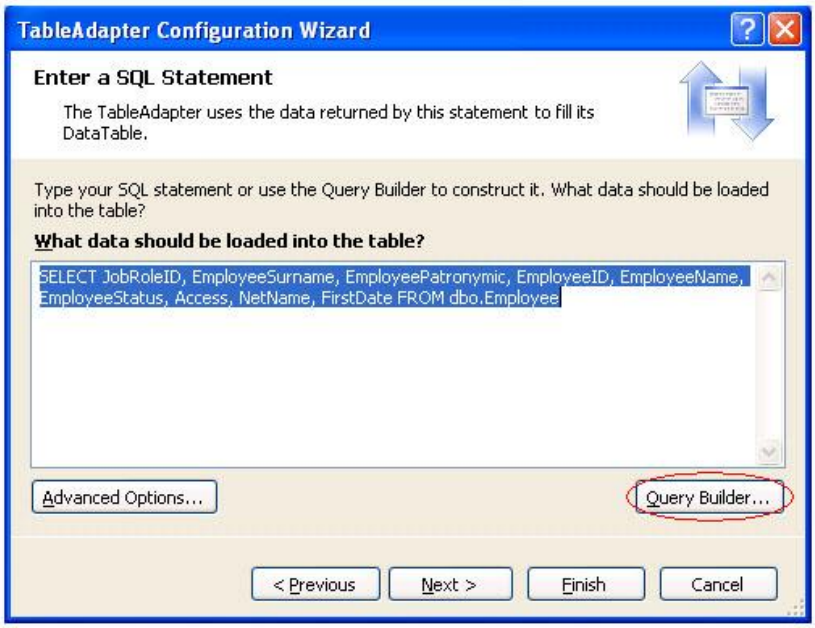

Рис. 6.3. Окно помощника конфигуратора адаптера

2. В окне *Table Adapter Configuration Wizard* нажмите кнопку *Query Builder* (рисунок 11.3) . В результате будет выведено окно *Query Builder* построителя *SQL* -запросов (рисунок 6.4).

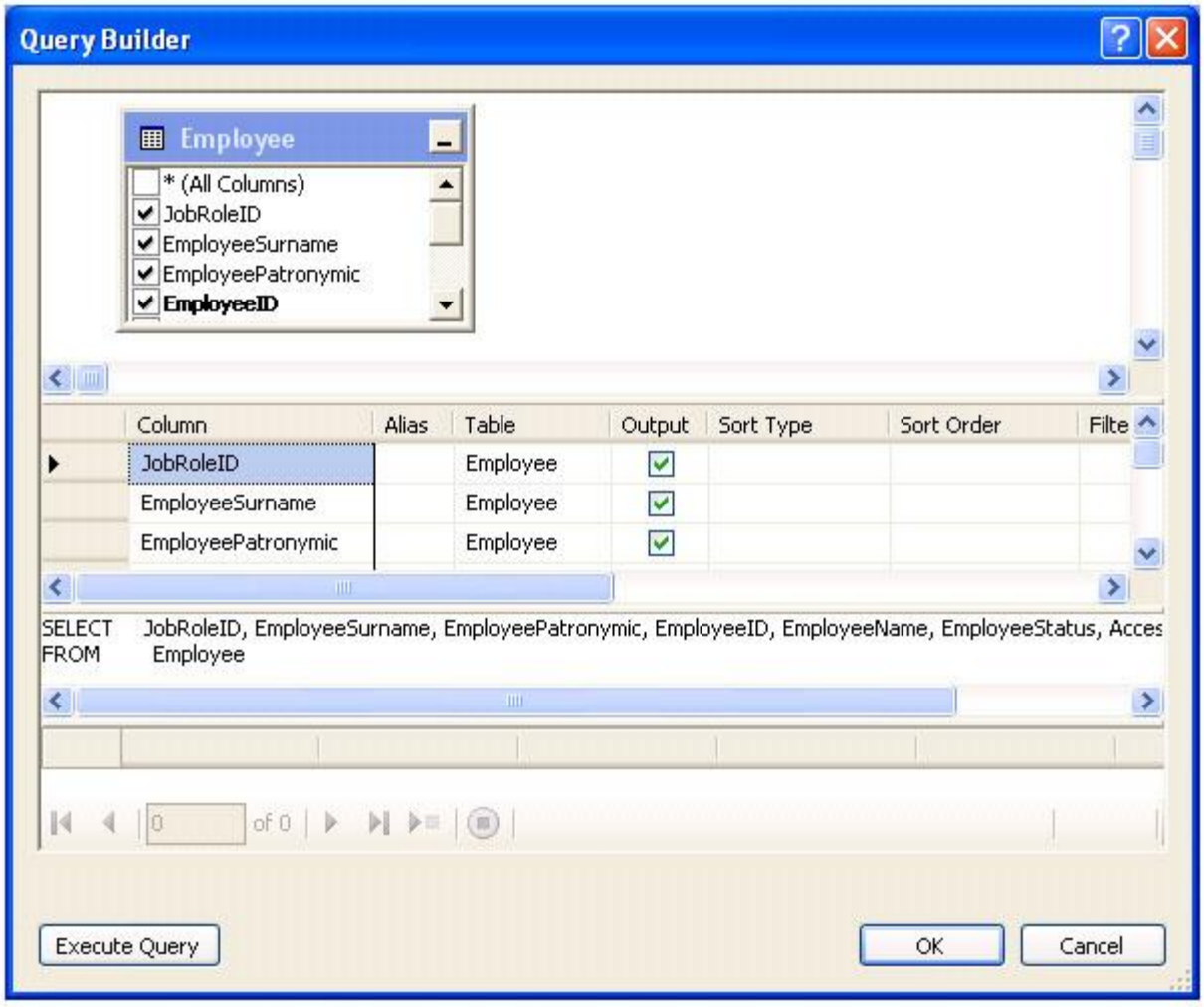

Рис. 6.4. Окно Query Builder построителя SQL-запросов

3. В окно *Query Builder* построителя *SQL* -запросов для колонки EmployeeSurname укажите тип сортировки (Sort Type) - по алфавиту - Ascending (рисунок 6.5). После этого в команде Select добавится код
ORDER BY EmployeeSurname
Полный вид команды Select следующий:
SELECT JobRoleID, EmployeeSurname, EmployeePatronymic,
EmployeeID, EmployeeName, EmployeeStatus, Access, NetName, FirstDate
FROM Employee
ORDER BY EmployeeSurname

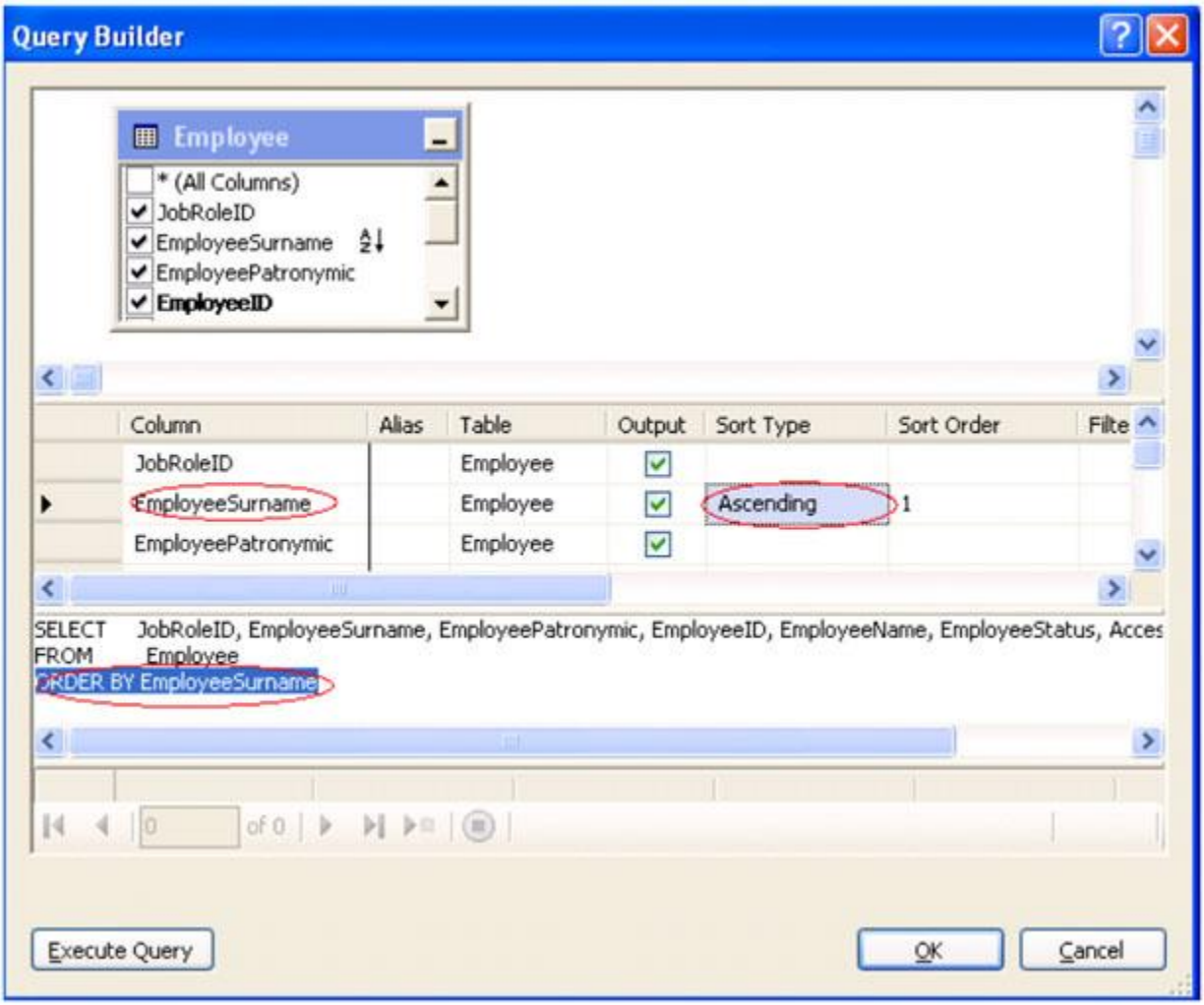

Рис. 6.5. Задание режима сортировки данных по фамилии сотрудника

4. Завершите формирование *SQL* -запроса, нажимая кнопки *"OK"* и *"Next"* в окне *Query Builder*.

После компиляции и запуска приложения список сотрудников будет упорядочен по алфавиту, как показано на рисунке 6.6.

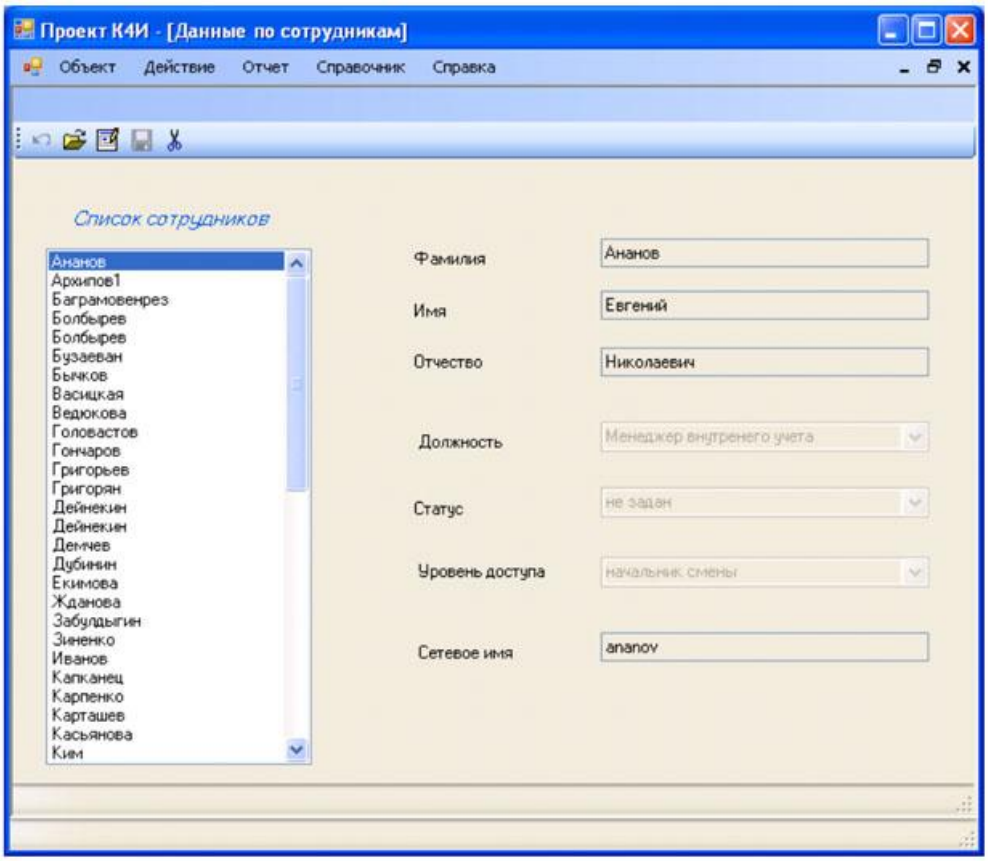

Рис. 6.6. Экранная форма с упорядоченным списком сотрудников

Формирование отображения списка сотрудников с полными данными

Для формирования в списке сотрудников информации об фамилии, имени и отчестве необходимо в таблицу Empoyee объектаdsEmployee класса DataSetEmployee добавить вычисляемый столбец и связать с этом столбцом *элемент списка* отображенияlistBoxEmployee.

Модификацию необходимо проводить в следующей последовательности.

1. Для элемента списка listBoxEmployee очистить привязку к источнику данных. Для этого в методе FormEmployee_Loadзакомментировать следующие строки кода:
2. //this.listBoxEmployee.DataSource = this.dsEmployee;
   //this.listBoxEmployee.DisplayMember = //"Employee.EmployeeSurname";
3. Добавляем в класс FormEmployee метод, который будет в таблицу Empoyee объекта dsEmployee класса *Data Set*добавлять вычисляемый столбец с фамилией, именем и отчеством сотрудника.
4. private void AddColumsFullName()
5. {
6. dsEmployee.Employee.Columns.Add("FullName", typeof(string),
7. "EmployeeSurname+' '+EmployeeName+' '+EmployeePatronymic");
   }
8. Добавляем в класс FormEmployee метод, который будет связывать элемент списка очистить привязку listBoxEmployee с объектом dsEmployee класса *Data Set*.
9. private void AddListBoxEmployeeDataSource()
10. {
11. listBoxEmployee.DataSource = this.dsEmployee;
12. listBoxEmployee.DisplayMember = "Employee.FullName";
    }

13. Вновь созданные методы будем вызывать при загрузке формы *FormEmployee*. Для этого модифицируем метод Load, добавив в него после метода EmployeeFill вызов методов AddColumsFullName и AddListBoxEmployeeDataSource.
14. После компилирования и запуска программы экранная форма "Учет сотрудников" должна иметь вид, аналогичный, приведенному на рисунке 6.7.

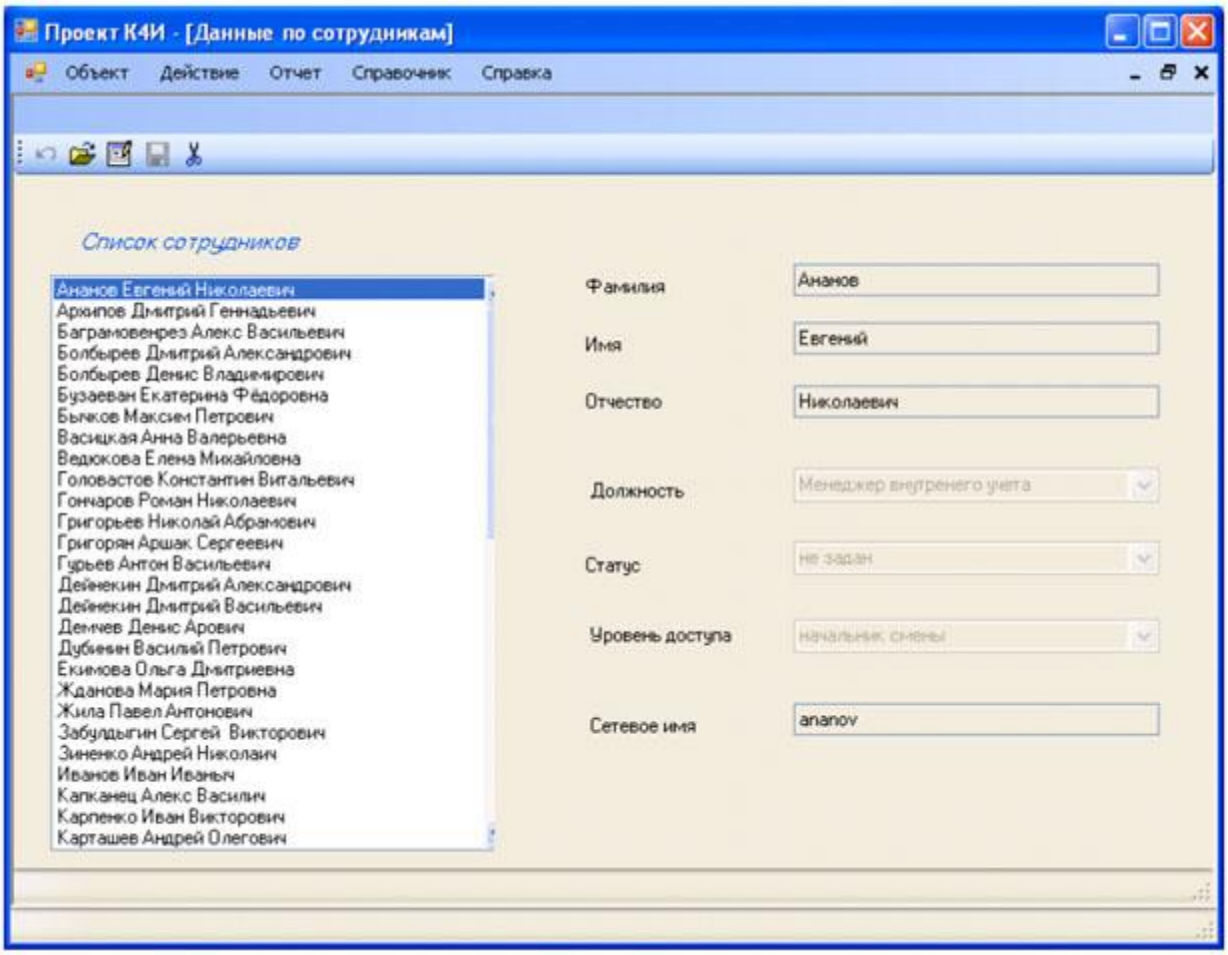

Рис. 6.7. Экранная форма с полными данными по сотрудникам

Задание на лабораторную работу

1. Изучите теоретический материал.
2. Модифицируйте приложение для обеспечения:
   - вывода списка сотрудников, содержащего фамилии, имена и отчества;
   - упорядочения выводимого списка сотрудников по алфавиту.
3. Протестируйте приложение.

Лабораторная работа №7 (6 ак.ч.)

Разработка приложения на базе WPF

Цель

В процессе выполнения работы необходимо разработать страничное/оконное WPF *приложение*.

Задание 1. Создать проект в соответствии с шаблоном "приложение WPF" и разработать интерфейс пользователя

Для разработки приложения необходимо создать WPF-проект (рис. 7.1). В окне " *Создать проект* " необходимо проверить установку библиотеки *.NET Framework 4* (1 – на рис. 7.1), выбрать шаблоны *Windows* (2 – на рис. 7.1), а среди имеющихся шаблонов задать*Приложение WPF* и в *поле* " *Имя* " ввести имя проекта *WpfApplProject*.

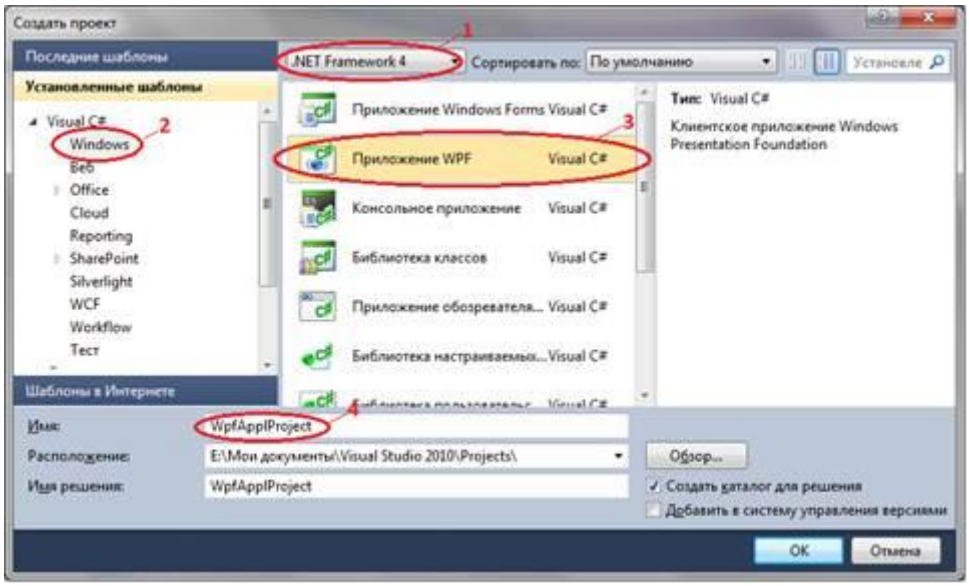

Рис. 7.1. Окно создания проекта

После нажатия кнопки "ОК" будет сформирован *шаблон* проекта. При этом инструментальная система сгенерирует следующий *XAML*-документ:

```
<Window x:Class="WpfApplProject.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Дизайнер проекта приведен на рис. 7.2.

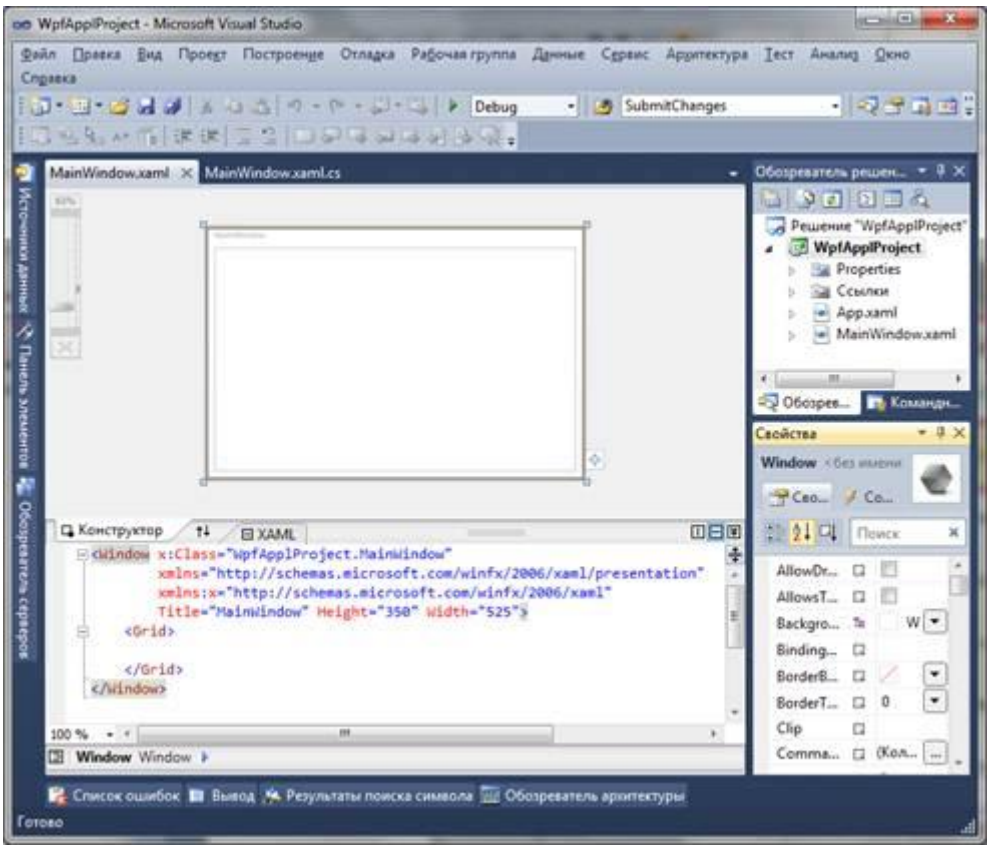

Рис. 7.2. Окно дизайнера проекта

В приведенном *XAML*-документе имеется один элемент верхнего уровня <*Window*>. *Дескриптор* </*Window*> завершает весь документ. В *XAML*-документе приведено *имя класса MainWindow*

x:Class="WpfApplProject.MainWindow

два пространства имен

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml

и три свойства:

Title="MainWindow" Height="350" Width="525"

Каждый *атрибут* соответствует определенному свойству класса *Window*. Приведенные атрибуты предписывают WPF создать окно с надписью *MainWindow* и размером 350x525 единиц. При компиляции и запуске проекта приложения на *дисплей* выводится окно, приведенное на рис. 7.3.

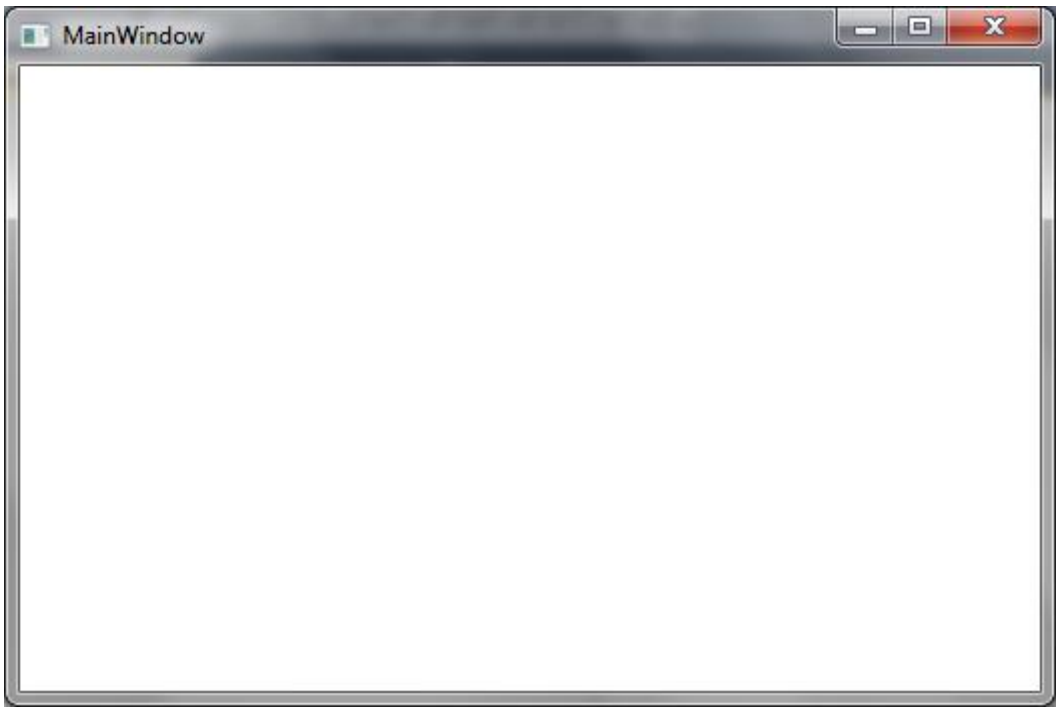

Рис. 7.3. Окно MainWindow проекта

Когда выполняется *компиляция* приложения, *XAML-файл*, который определяет пользовательский *интерфейс* (*MainWindow.xaml*), транслируется в объявление типа *CLR* , которое объединяется с логикой приложения из файла класса отдельного кода (*MainWindow.xaml.cs*).

Метод *InitializeComponent()* генерируется во *время компиляции* приложения и в исходном коде не присутствует.

Для программного управления элементами управления, описанными в *XAML*-документе, необходимо задать *XAML атрибут Name*. Так для задания имени элементу *Grid* необходимо записать следующую разметку:

```
<Grid Name="grid">
    </Grid>
```

Страницы приложения можно размещать внутри окон и внутри других страниц. В WPF при создании страничных приложений контейнером наивысшего уровня могут быть следующие объекты:

- *NavigationWindow*, который представляет собой несколько видоизмененную версию класса *Window* ;
- *Frame*, находящийся внутри другого окна или другой страницы;
- *Frame*, обслуживаемый непосредственно в *Internet Explorer*.

Для вставки страницы внутрь окна будем использовать *класс Frame* (рис. 7.4). Автоматически будет сгенерирован экземпляр класса*Frame* с фиксированными границами рис. 7.5).

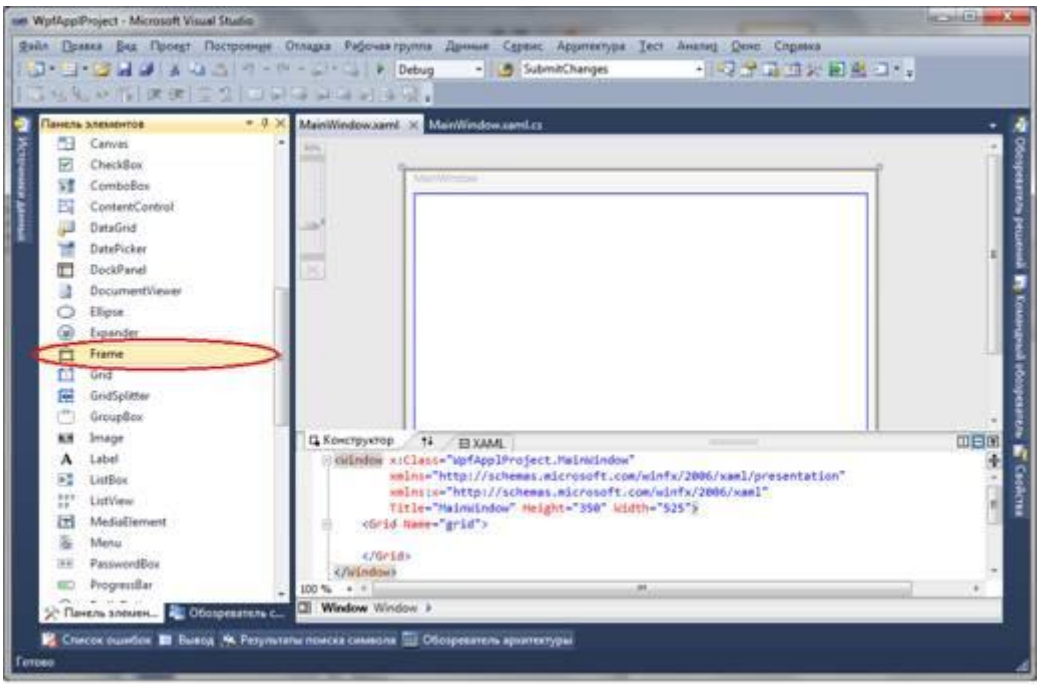

Рис. 7.4. Выбор класса Frame в панели инструментов

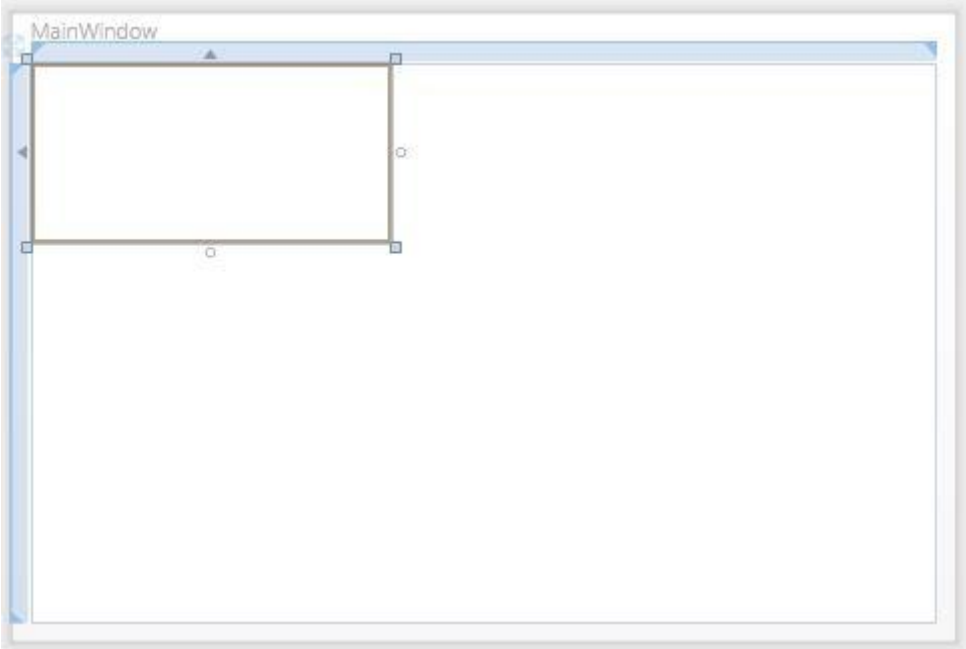

Рис. 7.5. Фрейм с фиксированными границами

В *XAML*-документ проекта будет добавлена следующая строка:

```
<Frame Height="100" HorizontalAlignment="Left"
Name="frame1" VerticalAlignment="Top" Width="200" />
```

С учетом того, что создается страничное *приложение* размеры фрейма не нужно фиксировать, поэтому изменим описание свойств фрейма:

```
<Frame Name="frame1" Margin="3" />
```

В результате *фрейм* заполнит всё окно (рис. 7.6):

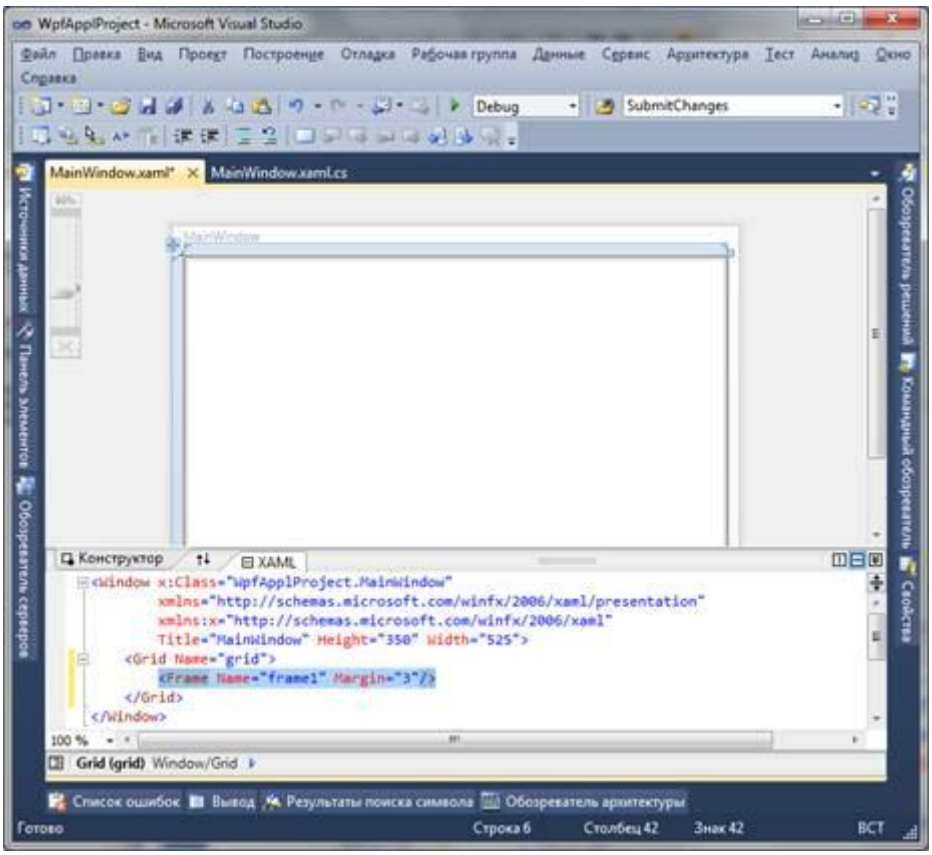

Рис. 7.6. Фрейм, заполняющий всё окно

Созданный *фрейм* необходим для размещения в нем страницы WPF-приложения. *Класс Page* допускает наличие только одного вложенного элемента. Он не является элементом управления содержимым и наследуется от класса *FrameworkElement*. *Класс Page*имеет небольшой набор дополнительных свойств, которые позволяют настраивать его внешний вид, взаимодействовать с контейнером и использовать навигацию. Для перехода на другую страницу необходимо использовать навигацию.

Добавьте в проект начальную страницу. Для этого в *Обозревателе решений* щелкните правой кнопкой мыши на проекте*WpfApplProject*. В контекстном *меню* выберите *пункт Добавить* (1 – на рис. 7.8), а в раскрывающимся *меню* - *пункт Страница* (2 нарис. 7.7).

В окне *Добавления нового элемента* необходимо выбрать *шаблон* " *Страница (WPF)* " (1) и задать имя страницы *PageMain* (2 на рис. 7.8).

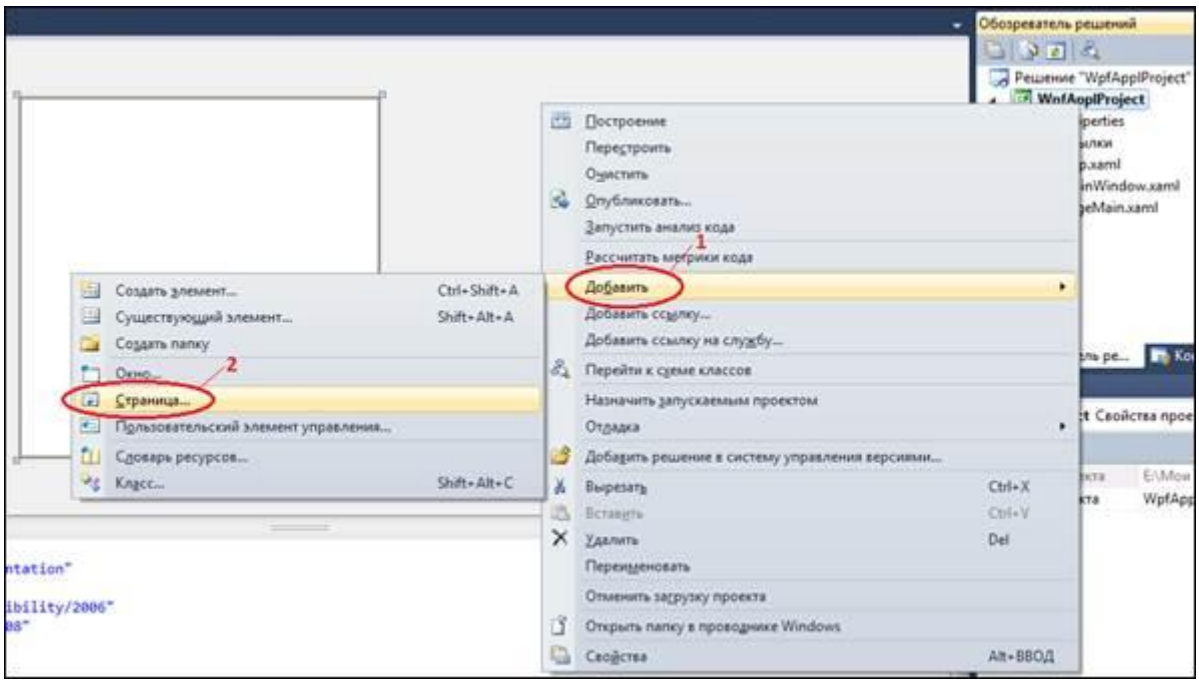

Рис. 7.7. Меню создания страницы

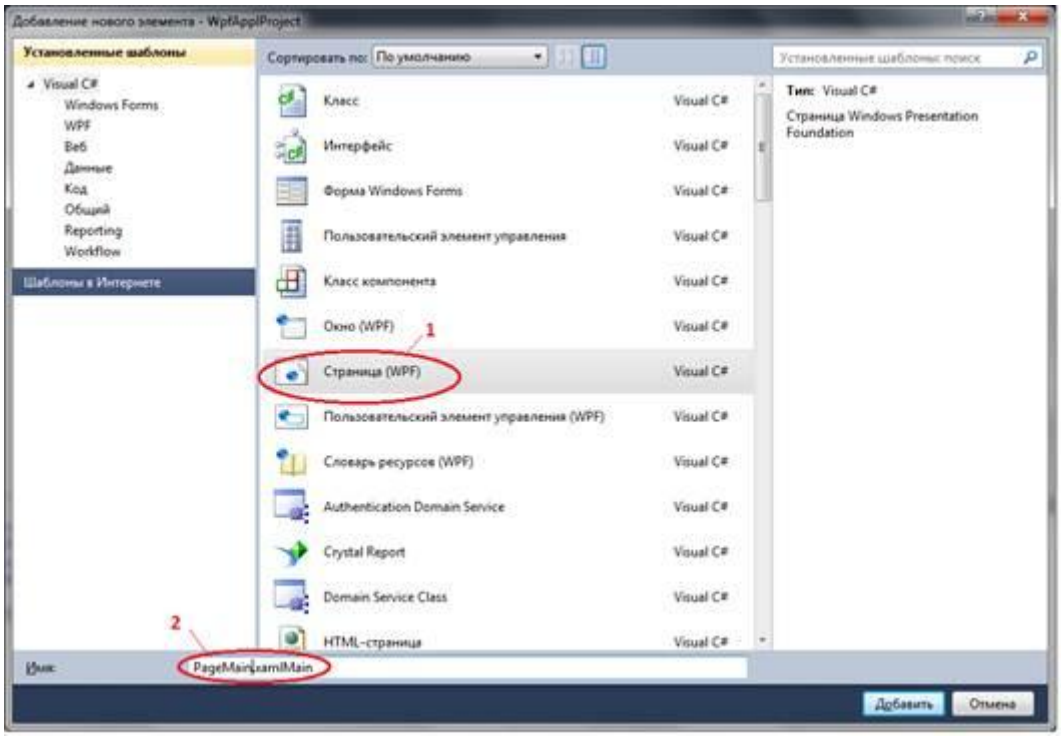

Рис. 7.8. Окно Добавления нового элемента

В дизайнере проекта сгенерируется страница *PageMain.xaml* (рис. 7.9).

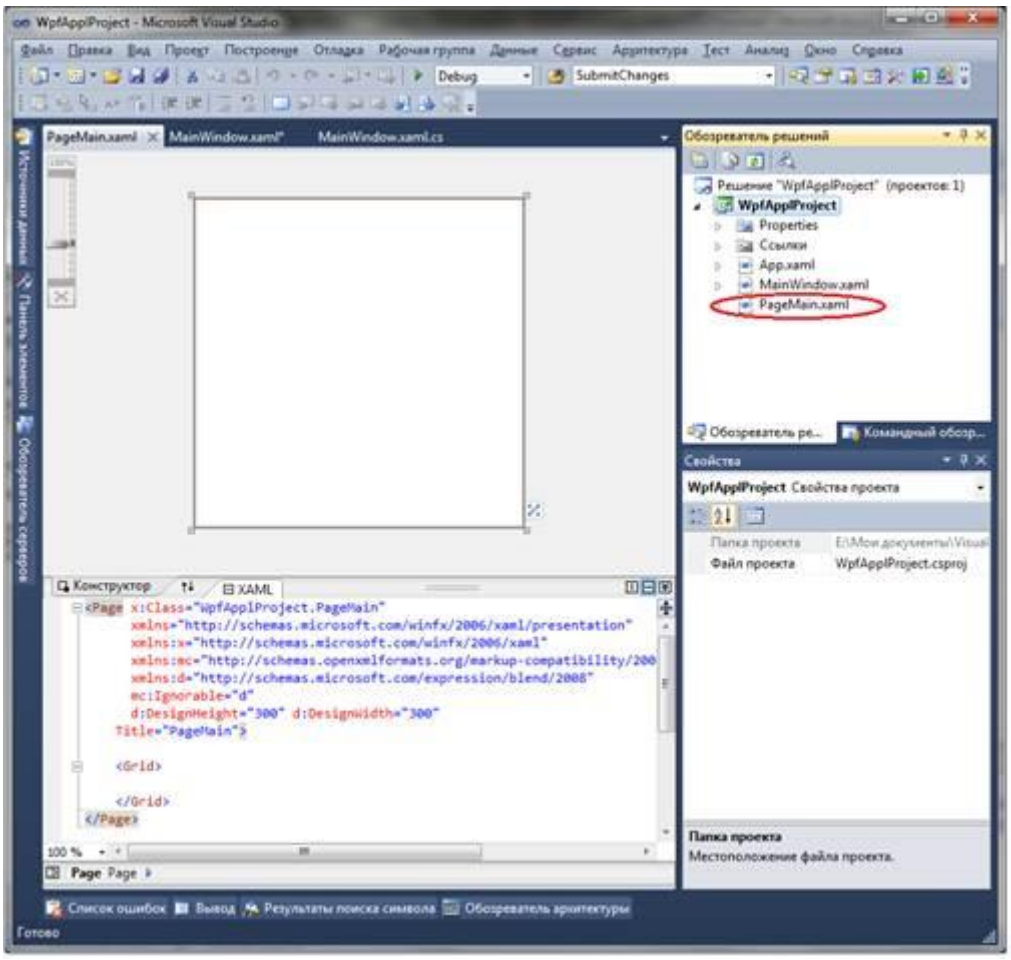

Рис. 7.9. Дизайнер страницы PageMain.xaml

В сгенерированной странице в качестве контейнера верхнего уровня используется *Grid*. Замените *Grid* на *контейнер StackPanel*.

Главная страница приложения в дизайнере представлена на рис. 7.10.

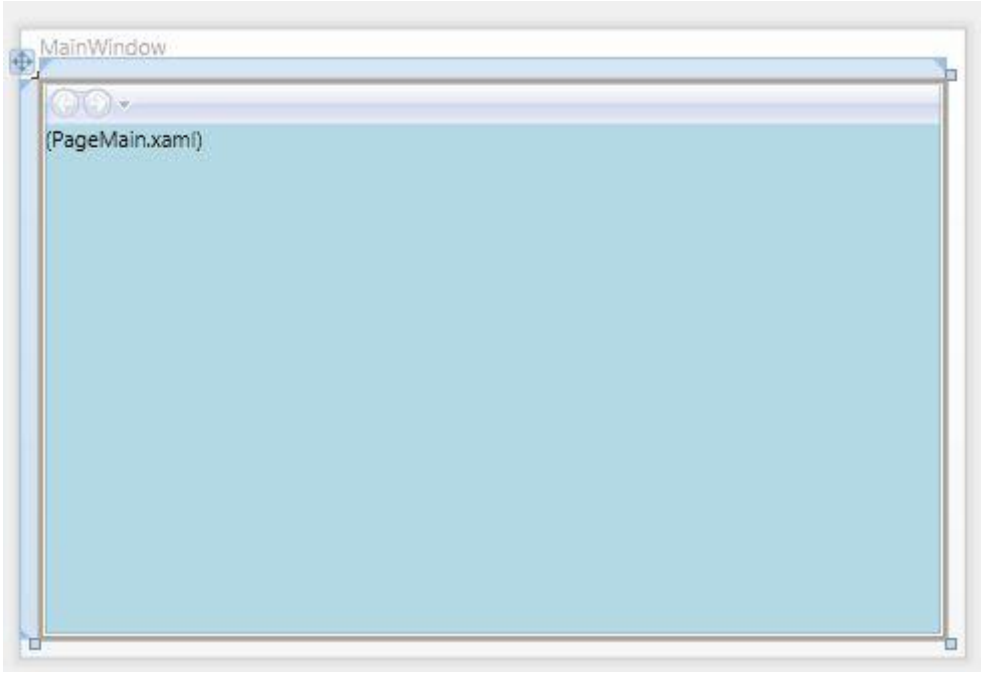

Рис. 7.10. Главная страница WPF-приложения в дизайнере

Измените свойство *Title* окна класса *Window*, в соответствии с вариантом лабораторной работы.

Основная страница должна обеспечивать переход на другие страницы, обеспечивающие инферфейс для отдельных функций и *выход*из системы. Для перехода на другие страцицы используйте гиперссылки. Гиперссылки позволяют пользователю перемещаться с одной страницы на другую. Элемент гиперссылки, соответствующий объекту класса *Hyperlink*, определяется следующей строкой:

```
<Hyperlink >Текст Гиперссылки</Hyperlink>
```

Класс Hyperlink имеет свойство *NavigateUri*. Данное свойство определяет на какую страницу будет переходить *приложение* при щелчке на соответствующей гиперссылке. Например, *NavigateUri="Page2.xaml"*.

В WPF гиперссылки являются не отдельными, а внутристроковыми потоковыми элементами, которые должны размещаться внутри другого поддерживающего их элемента. Это можно сделать, например в элементе *TextBlock*, который для гиперссылки является контейнером.

На первой странице создайте гиперссылки для перехода на страницы приложния, в соответствии в заданием.

Создание основного меню. Создайте основное *меню* с помощью класса *Menu*, который представляет *Windows элементы управления меню*, позволяющие иерархически организовать элементы, связанные с командами и обработчиками событий. *Меню*формируют из объектов *MenuItem* (имя пункта *меню*) и *Separator* (разделитель). *Класс MenuItem* имеет свойство *Header*, которое определяет текст элемента *меню*. Данный *класс* может хранить коллекцию объектов *MenuItem*, которая соответствует подпунктам*меню*. *Класс Separator* отображает горизонтальную линию, разделяющую пункты *меню*.

Добавьте в *StackPanel* страницы приложения *XAML*- описание *меню*, которое на верхнем уровне будет содержать, например два пункта *Действие* и *Отчет*. *Пункт Действие* включает подпункты *Отменить, Создать, Редактировать, Сохранить, Найти* и *Удалить*. Между пунктами *Отменить, Создать* и пунктами *Найти, Удалить* добавьте разделительные линии.

```
<Menu>
   <MenuItem Header="Действие" >
      <MenuItem Header="Отменить" ></MenuItem>
      <Separator></Separator>
      <MenuItem Header="Создать" ></MenuItem>
      <MenuItem Header="Редактировать" ></MenuItem>
      <MenuItem Header="Сохранить" ></MenuItem>
      <MenuItem Header="Найти" />
      <Separator></Separator>
      <MenuItem Header="Удалить" ></MenuItem>
   </MenuItem>
   <MenuItem Header="Отчет"></MenuItem>
</Menu>
```

Создание панели инструментов. *Панель инструментов* представляет специализированный *контейнер* для хранения коллекции элементов, обычно кнопок. Расположите в панели инструментов кнопки, функциональное назначение которых будет соответствовать подпунктам *меню Действие*, то есть *Отменить, Создать, Редактировать, Сохранить, Найти* и *Удалить*.

На лицевой стороне кнопок поместите графическое изображение соответствующего действия. Для этого добавьте в *файл* проекта папку *Images* и в неё включите графические объекты, которые можно найти в библиотеке графических объектов *Visual* Studio (*папка*VS2010ImageLibrary).

После добавления графических файлов в проект они будут отображены в обозревателе решений (рис. 7.11).

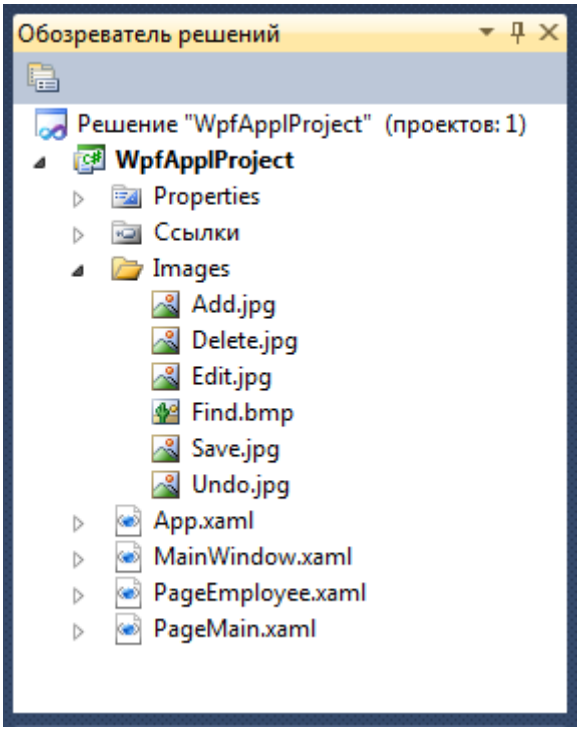

Рис. 7.11. Включение в проект папки Images с файлами рисунков

Для каждой кнопки задайте свойство *Name – имя объекта* в программе и свойство *ToolTip* с текстом всплывающей подсказки при наведении указателя мыши на кнопку.

Свойство *Margin* определяет *внешние отступы* для кнопки. Задание графического объекта для кнопки осуществляется определением для объекта *Image* источника *Source*, который должен соответствовать полному пути к графическому файлу.

```
<ToolBar Name="ToolBar1" Margin="3">
        <Button Name="Undo" ToolTip="Отменить редактирование/создание"
                Margin="5,2,5,2">
                <Image Source="Images/Undo.jpg" />
        </Button>
        ...
</ToolBar>
```

Проектирование интерфейсных элементов. При проектировании интерфейсных элементов для страницы приложения используются элементов контроля *ListBox, ListView, TextBox, TextBlock, ComboBox, DataGrid* и других.

Класс DataGrid представляет элемент управления, отображающий данные в настраиваемой сетке строк и столбцов. *По* умолчанию*DataGrid* автоматически создает столбцы, на основе источника данных. При этом генерируются следующие типы столбцов:

- *DataGridTextColomn* – для отображения в ячейках столбцов текстового содержимого;
- *DataGridCheckBoxColomn* – для отображения в ячейках столбцов логических данных;

- *DataGridComboBoxColomn* – для отображения в ячейках столбцов данных, когда имеется набор элементов для выбора;
- *DataGridHyperlinkColomn* – для отображения в ячейках столбцов элементов *Uri*.

Если разработчика не устраивает автоматическая генерация столбцов *DataGrid* можно её отключить. Для этого свойству*AutoGenerateColumns* необходимо присвоить *значение false*. Далее можно создать собственный набор столбцов (*Columns*), используя существующие типы столбцов или создать новый тип столбца с помощью шаблона *DataGridTemplateColumn*.

DataGrid поддерживает множество способов настройки отображения данных.

Задание 2. Разработать бизнес-логику приложения.

Для реализации функциональности приложения необходимо для страницы приложения установить механизм запуска задач при выборе соответствующих пунктов *меню* и нажатии на кнопки панели инструментов. Технология WPF предлагает модель команд для выполнения такой привязки.

Модель команд обеспечивает *делегирование* событий определенным командам и управление доступностью элементов управления в зависимости от состояния соответствующей команды. В WPF *команда* представляет собой задачу приложения и механизм *слежения*за тем, когда она может быть выполнена. В то же время сама *команда* не содержит конкретного кода выполнения задачи. Одна и та же *команда* может быть привязана к одному или нескольким интерфейсным элементам приложения. Инициируют команду источники, которые могут быть различными элементами управления, например пункты *меню MenuItem* или кнопки – *Button*. Целевым объектом команды является элемент, для которого предназначена эта *команда*.

Классы, реализующие команды должны поддерживать *интерфейс ICommand*. В этом интерфейсе определены два метода *Execute, CanExecute* и событие *CanExecuteChanged*.

```
public interface ICommand1
  {
    void Execute(object par);
    bool CanExecute(object par);
    event EventHandler CanExecuteChanged;
  }
```

Метод *Execute* может содержать код, реализующий бизнес-логику приложения. Метод *CanExecute* возвращает информацию о состоянии команды. Событие *CanExecuteChanged* вызывается при изменении состояния команды.

В WPF имеется библиотека базовых команд. Команды доступны через статические свойства следующих статических классов:

- *ApplicationCommands* ;
- *NavigationCommands* ;
- *EditingCommands* ;
- *MediaCommands*.

Для создания *пользовательских команд* целесообразно использовать классы *RoutedCommand*, который имеет *реализацию интерфейса ICommand*.

Для разработки пользовательских команд добавьте в проект папку *Commands* и в ней создайте *класс DataCommands*.

```
public class DataCommands
   {
      public static RoutedCommand Delete { get; set;}
      public static RoutedCommand Edit { get; set; }
      static DataCommands()
      {
         InputGestureCollection inputs = new InputGestureCollection();
         inputs.Add(new KeyGesture(Key.E, ModifierKeys.Control, "Ctrl+E"));
         edit = new RoutedCommand("Edit", typeof(DataCommands), inputs);
         inputs = new InputGestureCollection();
         inputs.Add(new KeyGesture(Key.D, ModifierKeys.Control, "Ctrl+D"));
         delete = new RoutedCommand("Delete", typeof(DataCommands), inputs);
      }
   }
```

В классе *DataCommands* объявлены два свойства *Delete* и *Edit* типа *RoutedCommand*. *Класс RoutedCommand* определяет команду, реализующую *ICommand*. В конструкторе данного класса определяется *объект inputs* типа *InputGestureCollection*. *КлассInputGestureCollection* представ ляет упорядоченную коллекцию объектов *InputGesture*, которые позволяют с помощью класса*KeyGesture* задать комбинацию клавиш для вызова команды.

Для использования страницей приложения пользовательских команд в *XAML*-документе необходимо добавить *пространство имен*, где расположен *класс DataCommands*. Данному пространству имен присвоим ссылку *command*.

```
xmlns:command="clr-namespace:WpfApplProject.Commands"
```

Теперь для страницы приложения сформируйте коллекцию объектов *CommandBinding*, которая осуществляет привязку команд для данного элемента и объявляет *связь* между командой, ее событиями и обработчиками.

```
<Page.CommandBindings>
            <CommandBinding Command="Undo"
             Executed="UndoCommandBinding_Executed"
      CanExecute="SaveCommandBinding_CanExecute" />
                        ...
</Page.CommandBindings>
```

Для класса *CommandBinding* свойство *Command* определяет ссылку на соответствующую команду, а свойства *Executed* и *CanExecute*задают обработчики событий при выполнении команды.

На странице приложения используются следующие команды: *Отменить, Создать, Редактировать, Поиск, Сохранить* и *Удалить*. Команды могут быть доступны или недоступны пользователю при работе приложения. Это проверяет метод *CanExecute* при генерации события *CanExecuteChanged*, которое вызывается при изменении состояния команды. Доступность команд определяется состоянием, в котором находится *приложение*. В тоже время выполнение какой-либо команды переводит, как правило, *приложение*в какое-либо другое состояние. Для проектируемого приложения можно определить следующие состояния:

- первоначальная загрузка страницы;

- просмотр данных;
- редактирование данных;
- создание новой записи.

В *код программы* класса страницы приложения введите логическое *поле isDirty* для управления доступностью команд.

```
private bool isDirty = true;
```

В код класса добавьте обработчики (реализация метода *Executed*), определяющие бизнес-логику приложения. На данном этапе проектирования системы обработчики будут содержать только *вывод* сообщений о вызове команды и изменение поля *isDirty*. Код обработчика команды *Удалить* приведен ниже.

```
private void UndoCommandBinding_Executed(object sender,
   ExecutedRoutedEventArgs e)
      {
         MessageBox.Show("Отмена");
         isDirty = true;
      }
```

В дальнейшем в обработчики необходима будет добавить код для обеспечения требуемой функциональности.

В коде класса необходимо добавить обработчики (реализация метода *CanExecute*), которые управляют доступностью команд.

```
private void EditCommandBinding_CanExecute(object sender, CanExecuteRoutedEventArgs e)
{
  e.CanExecute = isDirty;
}
private void SaveCommandBinding_CanExecute(object sender, CanExecuteRoutedEventArgs e)
{
  e.CanExecute = !isDirty;
}
```

Необходимо модифицировать *XAML*-документ в части задания свойства *Command* при описании пунктов *меню* и панели инструментов для привязки команд.

При описании *меню* (*Menu*) *XAML*-документ модифицирован следующим образом.

```
<Menu>
   <MenuItem Header="Действие" BorderThickness="3" >
      <MenuItem Header="Отменить" Command="Undo"></MenuItem>
            ...
</Menu>
```

Соответствующие изменения *XAML*-документа необходимо провести и для панели инструментов *ToolBar*.

```
<ToolBar Name="ToolBar1" Margin="3">
    <Button Name="Undo" Margin="5,2,5,2" Command="Undo"
          ToolTip="Отменить редактирование/создание" >
       <Image Source="Images/Undo.jpg">
      </Image>
    </Button>
...
```

</ToolBar>

Задание 3. Создать EDM-модель данных.

Для создания *EDM-модели данных* необходимо добавить в проект новый элемент – модель *ADO.NET EDM*, присвоив файлу модели имя с соответствии с заданием, в рассматриваемом примере модель имеет имя *TitleEmployee.edmx* (рис. 7.12).

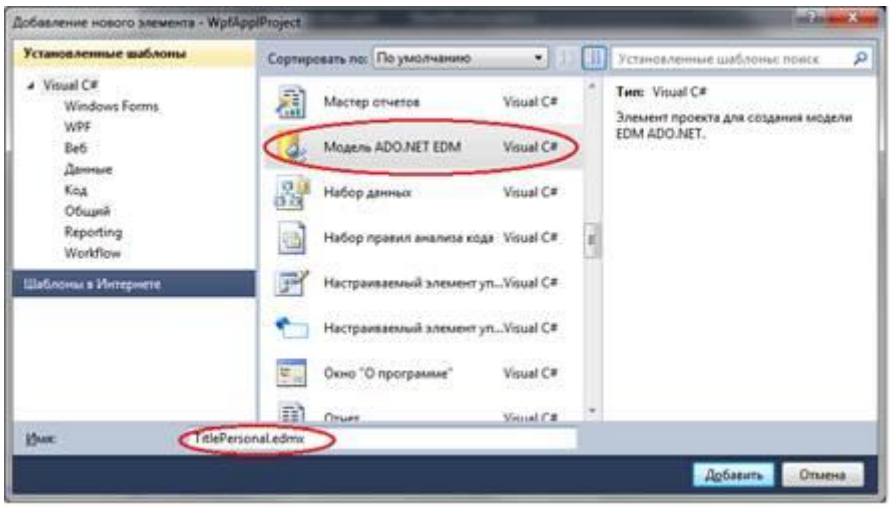

Рис. 7.12. Добавление в проект модели EDM

В мастере создания *EDM*-модели выберем опцию "Создать из *базы данных*" (рис. 7.13). Для создания соединения с базой данных в окне "Выбор подключения к данным" укажем соединение с базой данных (имя_сервера.имя_базы данных, в рассматриваемом примере – алексей-пк.TitlePerson) и зададим имя *модели данных - TitlePersonEntities* (рис. 7.14).

Рис. 6.13. Создание модели EDM из базы данных

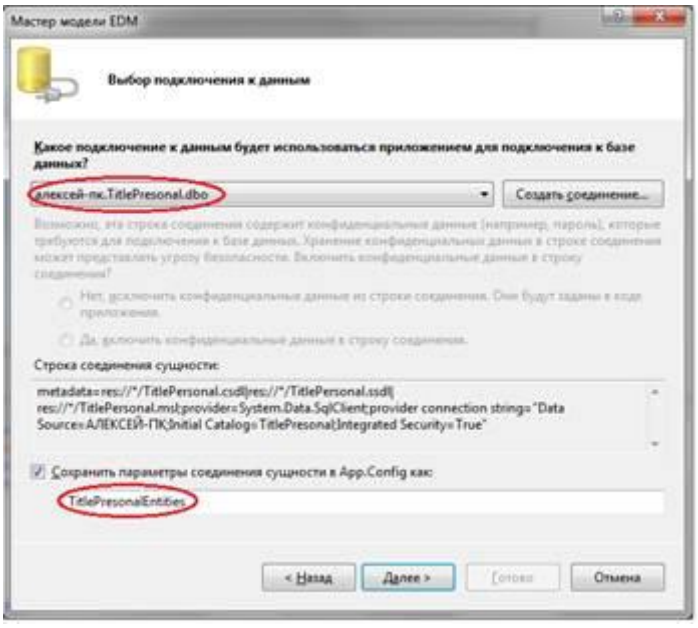

Рис. 7.14. Создание соединения с базой данных

В окне "Выбор *объектов базы данных*" отметим необходимые для приложения таблицы (в нашем случае это таблицы *Employee* и *Title*) и флаг формирования объектов в единственном или множественном числе (рис. 6.15).

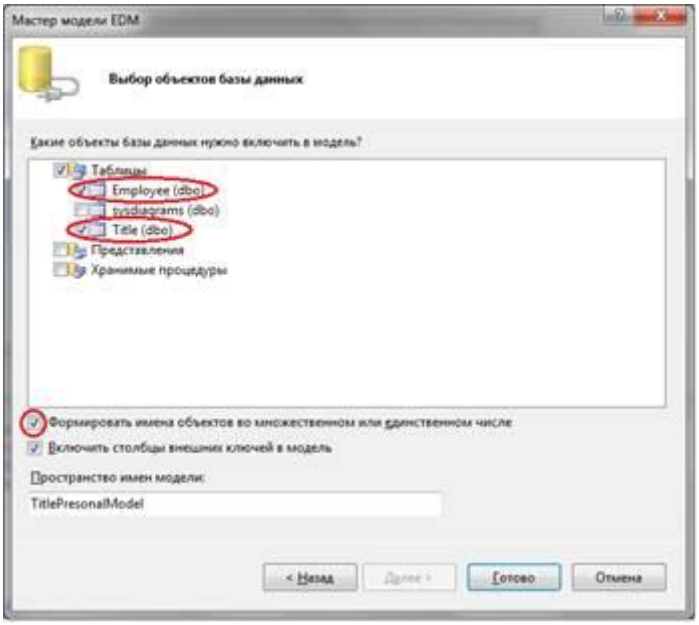

Рис. 7.15. Выбор таблиц базы данных

При завершении работы мастера создания *EDM*-модели в проект будет добавлен *файл* (в рассматриваемом примере *TitleEmployee.edmx* - рис. 7.16), ссыки на необходимые библиотеки и *конфигурационный файл*.

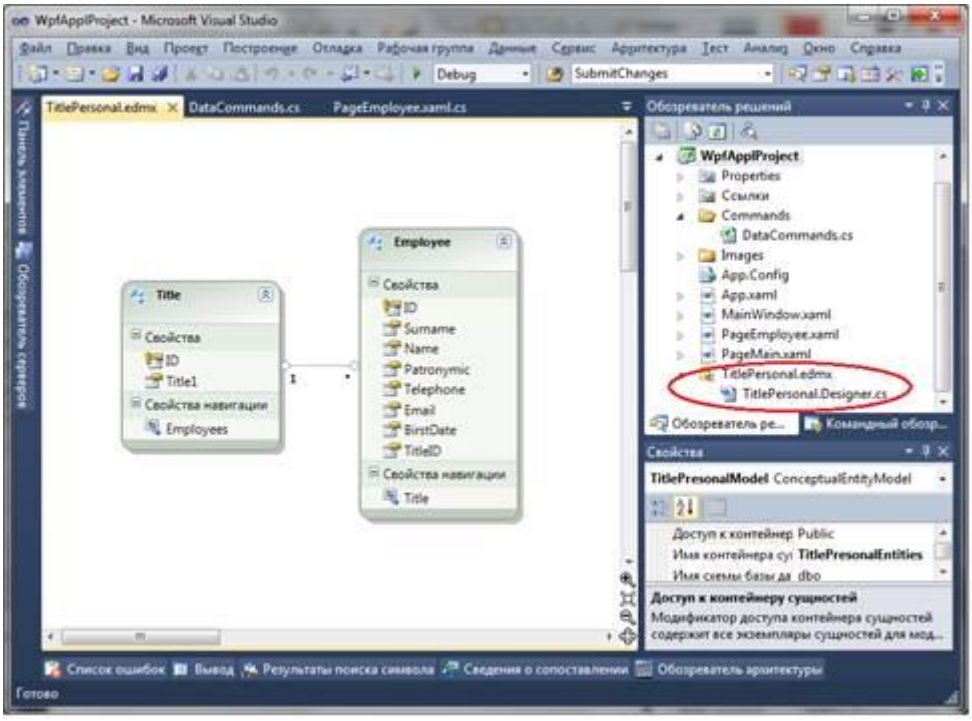

Рис. 7.16. Выбор таблиц базы данных

Автоматически сгенерированный *класс TitlePresonalEntities*, который наследуется от класса *ObjectContext*, представляет сущности *базы данныхTitlePerson*, содержит свойства, моделирующие таблицы *Employee* и *Title*, связи между таблицами.

При создании *модели данных* в проекте автоматически был сгенерирован *конфигурационный файл App.Config*, который содержит *строку соединения с базой данных*.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <connectionStrings>
    <add name="TitlePresonalEntities"
connectionString="metadata=res://*/TitlePersonal.csdl|res://*/TitlePersonal.ssdl|res://*
    /TitlePersonal.msl;provider=System.Data.SqlClient;provider connection string="Data
Source=АЛЕКСЕЙ-ПК;
    Initial Catalog=TitlePresonal;Integrated Security=True;MultipleActiveResultSets=True""
   providerName="System.Data.EntityClient" />
  </connectionStrings>
</configuration>
```

Задание 4. Произвести привязку данных к элементам контроля

Для взаимодействия приложения с базой данных необходимо в коде класса страницы объявить статическое свойство контекста данных сформированной *EDM*-модели. Это свойство целесообразно объявлять статическим.

Например:

```
public static TitlePresonalEntities DataEntitiesEmployee { get; set; }
```

Также необходимо объявить обобщенную коллекцию типа *ObservableCollection<Employee>* для работы приложения с коллекцией*объектов базы данных*. Этот тип представляет коллекцию динамических данных, обеспечивающих выдачу уведомления при получении и удалении элементов или при обновлении всего списка. Тип *ObservableCollection<T>* находится в пространстве имен*System.Collections.ObjectModel*, ссылку на которое нужно добавить в объявлении *класса приложения*.

```
using System.Collections.ObjectModel;
```

Экземпляры свойств контекста данных и коллекции необходимо создать в конструкторе класса страницы.

Например:

```
public PageEmployee()
{
    InitializeComponent();
   DataEntitiesEmployee = new TitlePresonalEntities();
   ListEmployee = new ObservableCollection<Employee>();
}
```

Формирование данных для приложения, которые должны предоставляться из *базы данных*, буде проводиться при загрузке страницы приложения. Для этого в *XAML*-документ *Page* добавьте свойство *Loaded*.

```
Loaded="Page_Loaded"
```

В код класса страницы приложения включаем обработчик *Page_Loaded*.

```
private void Page_Loaded(object sender, RoutedEventArgs e)
{
   ObjectQuery<Employee> employees = DataEntitiesEmployee.Employees;
   var queryEmployee = from employee in employees
```

```
                orderby employee.Surname
                select employee;
    foreach (Employee emp in queryEmployee)
    {
       ListEmployee.Add(emp);
    }
    DataGridEmployee.ItemsSource = ListEmployee;
}
```

Поле employees имеет тип ObjectQuery<Employee>. *Класс* ObjectQuery<T> представляет *запрос*, возвращающий коллекцию типизированных сущностей с любым количеством элементов. *Запрос* сформируем с помощью технологии *LINQ*.

```
var queryEmployee = from employee in employees
                orderby employee.Surname
                select employee;
```

Результаты запроса целесообразно отсортировать, например *по* фамилии сотрудника (*orderby employee.Surname*). Далее формируем коллекцию *объектов базы данных* и *источник данных* для сетки *DataGrid*.

```
foreach (Employee emp in queryEmployee)
    {
       ListEmployee.Add(emp);
    }
DataGridEmployee.ItemsSource = ListEmployee;
```

В результате проектирования в приложении сформирована коллекция с данными из таблицы *базы данных*.

Необходимо настроить сетку *DataGrid* для корректного отображения данных.

Модифицируйте общее описание *DataGrid*.

1. Определите привязку для источника данных.
   ItemsSource="{Binding}"
2. Отмените автоматическую генерацию столбцов.
   AutoGenerateColumns="False"
3. Установите привязку к левому краю страницы.
   HorizontalAlignment="Left"
4. Определите максимальные размеры сетки.
   MaxWidth="1000" MaxHeight="295"
5. Установите основной и альтернативный цвета заливки сетки.
6. RowBackground="#FFE6D3EF"
   AlternatingRowBackground="#FC96CFD4"
7. Определите цвет заливки и толщину линии для рамки сетки.
8. BorderBrush="#FF1F33EB"
   BorderThickness="3"
9. Определите высоту строк сетки.
   RowHeight="25"
10. Переопределите форму курсора при наведении указателя мыши на таблицу *DataGridEmployee*.

Модифицированное *XAML*-описание сетки *DataGrid*:

```
Cursor="Hand"
<DataGrid Name="DataGridEmployee"
ItemsSource="{Binding}"
AutoGenerateColumns="False"
HorizontalAlignment="Left"
MaxWidth="1000" MaxHeight="295"
RowBackground="#FFE6D3EF"
AlternatingRowBackground="#FC96CFD4"
BorderBrush="#FF1F33EB"
BorderThickness="3"
IsReadOnly="True"
RowHeight="25"
Cursor="Hand">
```

Привязка текстового поля

Для каждого текстового столбца задайте свойство привязки *Binding*. В расширении разметки привязки данного свойства определите свойство класса источника даннных, к которому привязывается колонка. Далее укажите режим двусторонней привязки (*Mode=TwoWay*), который определяет синхронное обновление как источника, так и приемника привязки. Способ обновления источника привязки задается свойством *UpdateSourceTrigger*, которое принимает значение *PropertyChanged*, что соответствует немедленному обновлению источника привязки каждый раз при изменении свойства цели привязки.

Например:

```
<DataGridTextColumn Header="Фамилия"
          Binding="{Binding Surname, Mode=TwoWay,
          UpdateSourceTrigger=PropertyChanged}"/>
```

Привязка выпадающего списка

Привязка данных к колонке типа *DataGridComboBoxColumn* требует определенной подготовительной работы. Если в таблице *модели данных* хранится не текстовое значение поля, а *внешний ключ* для другой таблицы, где находится данные, то в *EDM*-модели можно получить значение атрибута из связанных таблиц. Для этого используется атрибут связи в таблице *EDM*-модели.

Например, для обеспечения возможности работы с коллекцией таблицы *Title* в приложении добавте в проект папку *Model* и в ней создайте класс *ListTitle*.

```
public  class ListTitle: ObservableCollection<Title>
{
   public ListTitle()
   {
      ObjectQuery<Title> titles = PageEmployee.DataEntitiesEmployee.Titles;
      var queryTitle = from title in titles select title;
       foreach (Title titl in queryTitle)
      {
         this.Add(titl);
      }
   }
}
```

Класс *ListTitle* наследуется от класса обобщенной коллекции *ObservableCollection<T>* и его назначение создавать коллекцию объектов *Title*. Поле *titles* является запросом типа *ObjectQuery<Title>*. Данному полю присваивается свойство *Titles* контекста данных*DataEntitiesEmployee*, который определен в *классе приложения*.

Запрос LINQ получает данные из базы данных в поле *queryTitle* и затем в цикле *foreach* формируется коллекция класса *ListTitle*.

Для использования класса *ListTitle* в *XAML*-документе *класса приложения* необходимо объявить данный класс как ресурс. При этом нужно подключить пространство имен *WpfApplProject.Model*.

```
xmlns:core ="clr-namespace:WpfApplProject.Model"
```

Затем определите ресурс страницы с ключом *listTitle*.

```
<Page.Resources>
  <core:ListTitle x:Key="listTitle" />
</Page.Resources>
```

Далее модифицируйте *XAML*-описание для типа *DataGridComboBoxColumn*.

```
<DataGridComboBoxColumn Header="Должность"
      ItemsSource="{Binding Source={StaticResource listTitle}}"
      DisplayMemberPath="Title1"
      SelectedValueBinding="{Binding Path=TitleID, Mode=TwoWay,
      UpdateSourceTrigger=PropertyChanged}"
      SelectedValuePath="ID" />
```

Источник выпадающего списка задается как статический ресурс { *StaticResource listTitle* }. Выводимое в ячейки колонки поле должно соответствовать полю *Title1* таблицы *Title EDM*-модели (*DisplayMemberPath="Title1"*). Выбираемый в списке параметр (*SelectedValueBinding*) должен быть привязан к полю *TitleID* таблицы *Employee*.

```
SelectedValueBinding="{Binding Path=TitleID, Mode=TwoWay,
      UpdateSourceTrigger=PropertyChanged}"
```

Выбор в свойства *SelectedValueBinding* производится по пути определенному свойством *SelectedValuePath (SelectedValuePath="ID")*.

Привязка даты

При привязке даты ставится задача для невыделенной ячейки представлять дату в виде цифр дня, месяца и года, разделенных точками, а для выделенной ячейки – использовать класс для выбора даты.

Решение данной задачи осуществлено с помощью разработки двух шаблонов данных *DataTemplate*.

В первом шаблоне, для которого задан ключ *DateTemplate*, используется элемент управления *TextBlock*, свойство *Text* которого привязывается к атрибуту *BirstDate* таблицы *Employee*. Данные в привязке форматируются свойством *StringFormat* и строковые данные выравниваются по центру. Этот шаблон используется для визуализации данных в режиме их просмотра.

```
<DataTemplate x:Key="DateTemplate" >
   <TextBlock Text="{Binding BirstDate,
         StringFormat={}{0:dd\.}{0:MM\.}{0:yyyy}}"
         VerticalAlignment="Center" HorizontalAlignment="Center" />
</DataTemplate>
```

Второй шаблон с ключом *EditingDateTemplate* использует класс *DatePicker*. Этот шаблон используется в режиме редактирования данных.

```
<DataTemplate x:Key="EditingDateTemplate">
   <DatePicker SelectedDate="{Binding BirstDate, Mode=TwoWay,
    UpdateSourceTrigger=PropertyChanged}" />
</DataTemplate>
```

Разработанные шаблоны необходимо добавить к ресурсам в *XAML*-документ страницы *PageEmployee*.

Для настройки колонки "Дата рождения" модифицируем её *XAML*-описание.

```
<DataGridTemplateColumn Header="Дата рождения"
     CellTemplate="{StaticResource DateTemplate}"
     CellEditingTemplate="{StaticResource EditingDateTemplate}" />
```

Невыделенную ячейку колонки (*CellTemplate*) свяжите с ресурсом *DateTemplate*, а редактируемую ячейку (*CellEditingTemplate*) – с ресурсом *EditingDateTemplate*.

Ячейка столбца для отображения даты типа *DataGridTemplateColumn* имеет три представления (рис. 7.17). Если ячейка не выделена, то представление обычное текстовое (рис. 7.17а). При выделении ячейки прорисовывается свернутое содержание класса *DatePicker*(рис. 7.17б). При щелчке на ячейке появляется календарь (рис. 7.17в).

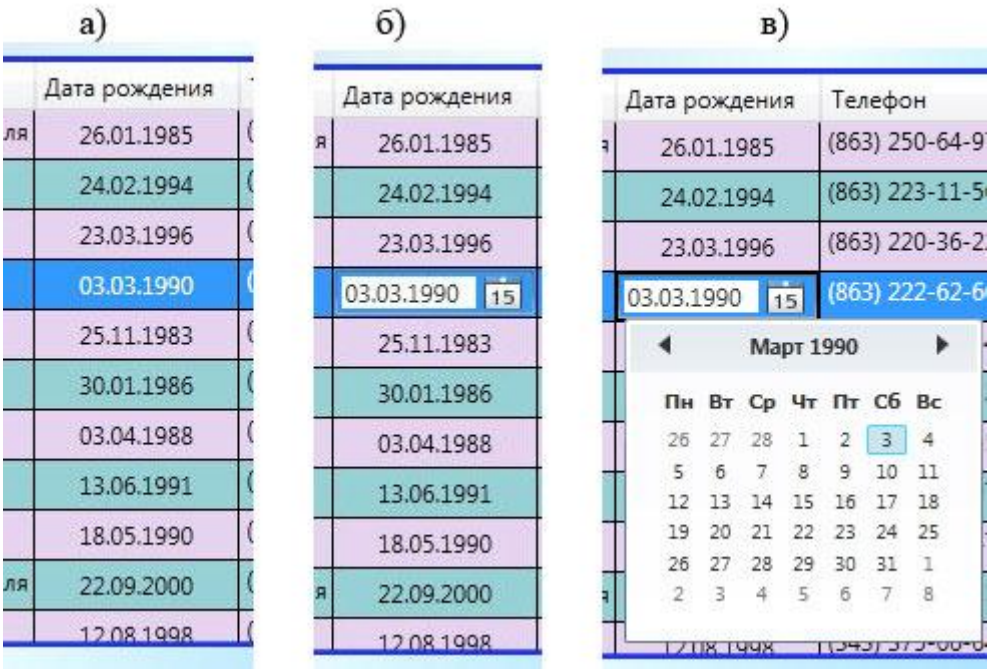

Рис. 7.17. Отображение даты в сетке DataGrid

Задание 5. Разработать методы манипулирования данными.

Для редактирования данных в приложении модифицируйте метод *EditCommandBinding_Executed*

```
private void EditCommandBinding_Executed(object sender,
ExecutedRoutedEventArgs e)
{
   DataGridEmployee.IsReadOnly = false;
   DataGridEmployee.BeginEdit();
   isDirty = false;
```

Свойству *IsReadOnly* сетки *DataGridEmployee* присвойте *значение false*, что обеспечит возможность редактирования строк и ячеек сетки. Далее используйте метод *BeginEdit* для начала редактирования ячейки, на которую наведен *указатель* мыши.

Результаты редактирования необходимо сохранить в базе данных. Для этого необходимо вызвать метод*SaveCommandBinding_Executed*.

```
private void SaveCommandBinding_Executed(object sender,
ExecutedRoutedEventArgs e)
{
   dataEntitiesEmployee.SaveChanges();
   isDirty = true;
   DataGridEmployee.IsReadOnly = true;
}
```

В методе *SaveCommandBinding_Executed* вызывается метод *SaveChanges* класса *dataEntitiesEmployee*. Метод *SaveChanges* сохраняет все обновления в источнике данных.

Для создания новых данных модифицируйте метод *NewCommandBinding_Executed*.

Например:

```
private void NewCommandBinding_Executed(object sender,
ExecutedRoutedEventArgs e)
{
   Employee employee = Employee.CreateEmployee(-1, "не задано", "не задано",
     "не задано", 0);
   employee.Telephone = "не задано";
   employee.Email = "не задано";
   try
   {
      DataEntitiesEmployee.Employees.AddObject(employee);
      ListEmployee.Add(employee);
      isDirty = false;
   }
   catch (DataServiceRequestException ex)
   {
      throw new ApplicationException(
            "Ошибка добавления данных" + ex.ToString());
   }
}
```

В методе *NewCommandBinding_Executed объект employee* класса *Employee* создается с помощью метода *CreateEmployee*. В блоке *try* ... *catch* созданный *объект employee* добавляется в *контекст* данных методом *AddObject* сущности *Employees EDM*-модели*DataEntitiesEmployee*, а также в коллекцию *ListEmployee*.

При инициализации команды создания данных в сетке формируется первоначальная строка с данными "*по* умолчанию". В сформированную строку необходимо ввести реальные данные сохранить их, вызвав команду "Сохранить".

Для удаления данных *по* сотруднику используем метод *DeleteCommandBinding_Executed*.

```
private void DeleteCommandBinding_Executed(object sender,
            ExecutedRoutedEventArgs e)
{
   Employee emp = DataGridEmployee.SelectedItem as Employee;
   if (emp != null)
   {
      MessageBoxResult result = MessageBox.Show("Удалить данные ",
"Предупреждение", MessageBoxButton.OKCancel);
      if (result == MessageBoxResult.OK)
      {
         DataEntitiesEmployee.DeleteObject(emp);
         DataGridEmployee.SelectedIndex =
          DataGridEmployee.SelectedIndex == 0 ? 1 : DataGridEmployee.SelectedIndex - 1;
         ListEmployee.Remove(emp);
         DataEntitiesEmployee.SaveChanges();
      }
   }
   else
   {
      MessageBox.Show("Выберите строку для удаления");
   }
}
```

В методе определяется *объект emp* класса *Employee*, который выделен в сетке *DataGridEmployee*.

Если *объект emp* найден, то с помощью диалогового окна класса *MessageBoxResult* выводится сообщение с данными, которые предполагается удалить.

При подтверждении *операции* удаления *объект emp* удаляется из контекста методом *DeleteObject*.

```
DataEntitiesEmployee.DeleteObject(emp);
```

Объект emp удаляется из коллекции *ListEmployee*.

```
ListEmployee.Remove(emp);
```

Изменения вносятся в базу данных.

```
DataEntitiesEmployee.SaveChanges();
```

Метод *UndoCommandBinding_Executed* реализует отмену редактирования последних элементов сетки *DataGridEmployee* и переводит её в режим просмотра.

Для реализации данной функциональности необходимо провести изменения в коде приложения. Для метода *Page_Loaded* основной код выделите в метод *GetEmployees*.

```
private void GetEmployees()
{
   ObjectQuery<Employee> employees = DataEntitiesEmployee.Employees;
   var queryEmployee = from employee in employees
                       orderby employee.Surname
                       select employee;
   foreach (Employee emp in queryEmployee)
   {
```

```
        ListEmployee.Add(emp);
    }
    DataGridEmployee.ItemsSource = ListEmployee;
}
```

С учетом созданного метода *GetEmployees* метод *Page_Loaded* будет иметь следующий вид.

```
private void Page_Loaded(object sender, RoutedEventArgs e)
{
    GetEmployees();
    DataGridEmployee.SelectedIndex = 0;
}
```

Создайте метод *RewriteEmployee*, который будет обновлять *контекст* данных и коллекцию *ListEmployee*.

```
private void RewriteEmployee()
{
    DataEntitiesEmployee = new TitlePresonalEntities();
    ListEmployee.Clear();
    GetEmployees();
}
```

С учетом проведенных модификаций кода метод *UndoCommandBinding_Executed* будет иметь следующий вид.

```
private void UndoCommandBinding_Executed(object sender,
ExecutedRoutedEventArgs e)
{
    RewriteEmployee();
    DataGridEmployee.IsReadOnly = true;
    isDirty = true;
}
```

Задание 6. Реализовать валидацию данных

Разработку пользовательского правила *валидации* данных рассмотрим на примере проверки шаблона при вводе свойства *Email*, то есть адреса электронной почты. В строке ввода должны присутствовать символы @ и точка.

Спроектируйте *правило проверки* для привязки, которое будет использоваться со столбцом "Электронная *почта*". Для этого необходимо создать *класс* правила проверки *EmailRule*, который должен наследоваться от базового класса *ValidationRule*. *КлассEmailRule* поместите во вновь созданную папку *ValidationRules* проекта.

```
public class EmailRule: ValidationRule
{
    public override ValidationResult Validate(object value,
                System.Globalization.CultureInfo cultureInfo)
    {
        string email = string.Empty;
        if (value != null)
        {
            email = value.ToString();
```

```
        }
        else
            return new ValidationResult(false, " Адрес электронной почты не задан! ");
        if (email.Contains("@") && email.Contains("."))
        {
            return new ValidationResult(true, null);
        }
        else
        {
            return new ValidationResult(false,
"Адрес электронной почты должен содержать
символы @ и (.) точки \n Шаблон адреса:   adres@mymail.com");
        }

    }
}
```

В классе правил проверки необходимо переопределить метод *Validate*, который возвращает результат проверки – экземпляр класса*ValidationResult*. Если проверка проходит успешно, тогда возвращаемым значением является *объект* класса *ValidationResult*, который создается с параметрами *true* и *null*.

```
return new ValidationResult(true, null);
```

Если проверка приводит к выявлению несоответствия, то *класс ValidationResult*, создается с параметрами *false* и строка сообщения об ошибке.

```
return new ValidationResult(false, "Адрес электронной почты должен содержать
символы @ и (.) точки \n Шаблон адреса:   adres@mymail.com");
```

Для использования объекта *EmailRule* в *XAML*-документе страницы приложения добавьте в её описание *пространство имен WpfApplProject.ValidationRules*.

```
xmlns:rule="clr-namespace:WpfApplProject.ValidationRules"
```

Внесите изменения в *XAML*-описание колонки *Электронная почта*.

```
<DataGridTextColumn  Header="Электронная почта" EditingElementStyle="{StaticResource
errorStyle}">
   <DataGridTextColumn.Binding >
      <Binding Path="Email" Mode="TwoWay" UpdateSourceTrigger="PropertyChanged"
ValidatesOnExceptions ="True" >
         <Binding.ValidationRules>
            <rule:EmailRule />
        </Binding.ValidationRules>
      </Binding>
   </DataGridTextColumn.Binding>
</DataGridTextColumn>
```

Для привязки *Binding* свойству *ValidatesOnExceptions* задайте *значение true*. Задание данного свойства обеспечивает использование элемента *ExceptionValidationRule*. *ExceptionValidationRule* предоставляет встроенное *правило проверки*, проверяющее возникновение исключений при обновлении свойства источника. В случае возникновения ошибки механизм привязки создает *ValidationError* с исключением и

добавляет его в коллекцию *Validation.Errors* привязанного элемента. Отсутствие ошибок сбрасывает это состояние обратной связи проверки, если другое правило не вызовет событие проверки.

Правило проверки задается для свойства *ValidationRules* класса *Binding*.

```
<Binding.ValidationRules>
        <rule:EmailRule />
</Binding.ValidationRules>
```

Модель *привязки данных* WPF позволяет связывать *ValidationRules* с объектом *Binding*, используя пользовательские правила. Подсистема привязки проверяет каждое из *ValidationRule*, связанных с привязкой, каждый раз, когда вводимое *значение* (*значение*свойства цель привязки) переносится в свойство источник привязки.

В WPF имеются стандартные стили для отображения ячеек сетки при возникновении ошибки ввода, но они являются недостаточно информативными. Для более эффектного выделения ячейки сетки при ошибке ввода создайте специальный стиль *errorStyle*.

```
<Style x:Key="errorStyle" TargetType="{x:Type TextBox}">
  <Setter Property="Padding" Value="-2"/>
  <Style.Triggers>
    <Trigger Property="Validation.HasError" Value="True">
      <Setter Property="Background" Value="Red"/>
      <Setter Property="BorderThickness" Value="1" />
      <Setter Property="ToolTip"
          Value="{Binding RelativeSource={RelativeSource Self},
        Path=(Validation.Errors)[0].ErrorContent}"/>
    </Trigger>
  </Style.Triggers>
</Style>
```

В стиле *errorStyle триггер* срабатывает, когда свойство *Validation.HasError* принимает *значение true*. При этом цвет заливки ячейки становится красным и при наведении на ячейку указателя мыши выводится сообщение-подсказка (свойство *ToolTip*), текст которого определяется в правиле проверки *EmailRule* при обнаружении ошибки (*Validation.Errors*)[0] .*ErrorContent*).

Стандартные средства WPF при обнаружении ошибки ввода в ячейке помечают строку сетки красным восклицательным знаком. Создадим *шаблон ControlTemplate* для визуального указания ошибки при проверке строки. Данный *шаблон* задается для свойства*RowValidationErrorTemplate* сетки *DataGrid*. Проектируемый *шаблон* должен обеспечить *вывод* красного круга с белым восклицательным знаком слева от строки сетки с обнаруженной ошибкой ввода. При наведении указателя мыши на круг должна выводиться подсказка, сформированная в классе правил проверки ввода.

```
<DataGrid.RowValidationErrorTemplate>
  <ControlTemplate>
    <Grid Margin="0,-2,0,-2"
        ToolTip="{Binding RelativeSource={RelativeSource FindAncestor,
                        AncestorType={x:Type DataGridRow}},
      Path=(Validation.Errors)[0].ErrorContent}">
```

```
        <Ellipse StrokeThickness="0" Fill="Red" Width="{TemplateBinding FontSize}"
                                    Height="{TemplateBinding FontSize}" />
        <TextBlock Text="!" FontSize="{TemplateBinding FontSize}" FontWeight="Bold"
                                Foreground="White" HorizontalAlignment="Center"  />
    </Grid>
  </ControlTemplate>
</DataGrid.RowValidationErrorTemplate>
```

Задание 7. Разработать методы поиска данных.

Функцию поиска данных в приложений необходимо выполнить на основе данных, имеющихся в базе данных варианта лабораторной работы. параметры поиска необходимо согласовать с преподавателем.

Общий подход к поиску реализации поиска данных, получаемых из *базы данных*, рассмотрим на примере поиска сотрудника *по*фамилии и должности. Для реализации функций поиска добавьте на страницу приложения элементы контроля в соответствии с рис. 7.18.

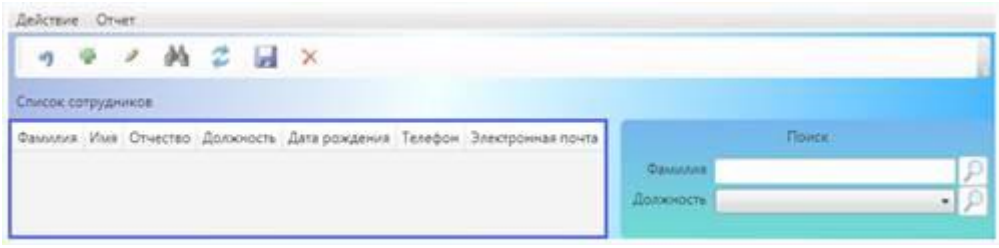

Рис. 7.18. Дизайн страницы с элементами контроля для поиска

XAML-описание элементов контроля, поддерживающих *поиск* имеет следующий вид.

```
<TextBlock Name="TextBlockSurname" Text="Фамилия" />
<TextBlock Name="TextBlockTitle" Text="Должность" />
<TextBox Name="TextBoxSurname" TextChanged="TextBoxSurname_TextChanged"/>
<ComboBox Name="ComboBoxTitle
ItemsSource="{Binding Source={StaticResource listTitle}}"
DisplayMemberPath="Title1"
SelectionChanged="ComboBoxTitle_SelectionChanged"/>
<Button Name="ButtonFindSurname" ToolTip="Поиск по фамилии"
IsEnabled="False" Click="ButtonFindSurname_Click">
<Image Source="Images/Search.jpg"/>
</Button>
<Button Name="ButtonFindTitle" ToolTip="Поиск по должности"
IsEnabled="False" Click="ButtonFindTitle_Click">
<Image Source="Images/Search.jpg" />
</Button>
```

Элементы контроля размещаем в рамке *BorderFind*. В рамке располагаем сетку *gridFind* с тремя колонками и строками. Первая строка в объединенных колонках содержит текстовый блок *find* с текстом "*Поиск*". Во второй и третьей строках первой колонки размещены текстовые блоки *TextBlockSurname* и *TextBlockTitle* соответственно. Во второй колонке размещаются элементы контроля для ввода информации: текстовый блок *TextBoxSurname* (вторая строка) и выпадающий *список ComboBoxTitle* (третья строка). В

третьей колонке размещаются кнопки: для поиска *по* фамилии *ButtonFindSurname* (вторая строка) и для поиска *по* должности *ButtonFindTitle*(третья строка).

При загрузке страницы *PageEmployee* рамка *BorderFind* невидима, так как свойству *Visibility* задано *значение "Hidden"*. При активизации команды *Find* из *меню* или панели инструментов запускается обработчик *FindCommandBinding_Executed*, который делает рамку *ButtonFindTitle* видимой.

```
private void FindCommandBinding_Executed(object sender,
ExecutedRoutedEventArgs e)
      {
        BorderFind.Visibility = System.Windows.Visibility.Visible;
      }
```

Если в текстовом блоке *TextBoxSurname* не введена *информация* или в выпадающем списке *ComboBoxTitle* не сделан выбор, то кнопки *ButtonFindSurname* и *ButtonFindTitle* будут недоступны.

При вводе информации в текстовый блок *TextBoxSurname* запускается обработчик *TextBoxSurname_TextChanged*, который делает доступной кнопку *ButtonFindSurname*.

```
private void TextBoxSurname_TextChanged(object sender, TextChangedEventArgs e)
{
   ButtonFindSurname.IsEnabled = true;
   ButtonFindTitle.IsEnabled = false;
   ComboBoxTitle.SelectedIndex = -1;
}
```

При нажатии кнопки *ButtonFindSurname* запускается обработчик *ButtonFindSurname_Click*.

```
private void ButtonFindSurname_Click(object sender, RoutedEventArgs e)
{
   string surname = TextBoxSurname.Text;
   DataEntitiesEmployee = new TitlePresonalEntities();
   ListEmployee.Clear();
   ObjectQuery<Employee> employees = DataEntitiesEmployee.Employees;
   var queryEmployee = from employee in employees
                       where employee.Surname == surname
                       select employee;
   foreach (Employee emp in queryEmployee)
   {
      ListEmployee.Add(emp);
   }
   if (ListEmployee.Count > 0)
   {
      DataGridEmployee.ItemsSource = ListEmployee;
      ButtonFindSurname.IsEnabled = true;
      ButtonFindTitle.IsEnabled = false;
   }
   else
      MessageBox.Show("Сотрудник с фамилией \n"+surname+"\n не найдан",
```

```
            "Предупреждение", MessageBoxButton.OK, MessageBoxImage.Warning);
}
```

В данном обработчике события *Click* выполняется *LINQ запрос* на получение данных из базы *TitlePersonal* в соответствии с заданной фамилией сотрудника.

```
var queryEmployee = from employee in employees
                    where employee.Surname == surname
                    select employee;
```

При выполнении запроса *по* фамилии страница *PageEmployee* имеет вид, приведенный на рис. 7.19

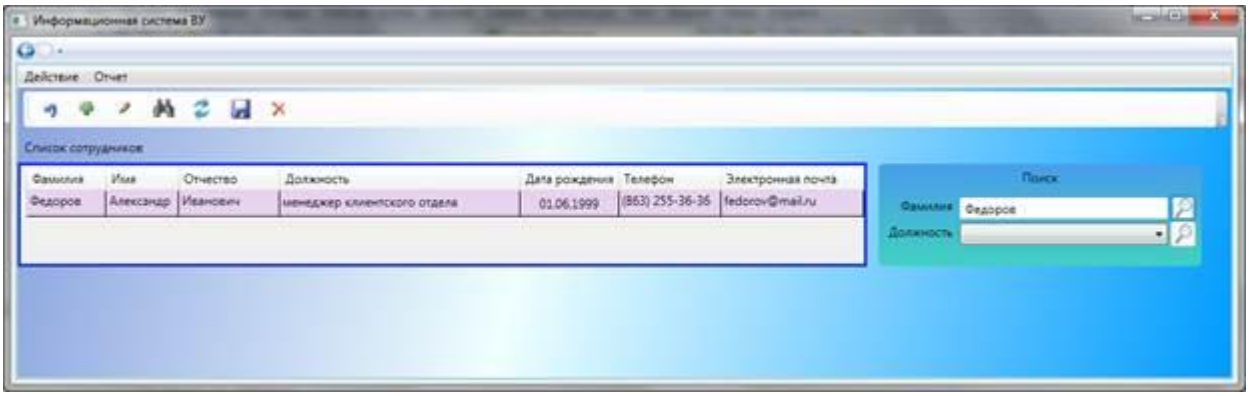

Рис. 7.19. Поиск по фамилии

Найденные данные *по* сотруднику можно редактировать и удалять.

Если *поиск в базе данных* не привел к нахождению информации, то выдается *информационное сообщение* (рис. 7.20).

Рис. 7.20. Неудачный поиск по фамилии

Для поиска информации из *базы данных по* должности сотрудников необходимо сделать выбор из выпадающего списка*ComboBoxTitle*. В результате выбора срабатывает обработчик *ComboBoxTitle_SelectionChanged*, который делает доступной кнопку*ButtonFindTitle*.

```
private void ComboBoxTitle_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
   ButtonFindTitle.IsEnabled = true;
   ButtonFindSurname.IsEnabled = false;
   TextBoxSurname.Text = "";
}
```

При нажатии на кнопку *ButtonFindTitle* срабатывает обработчик *ButtonFindTitle_Click*.

```
private void ButtonFindTitle_Click(object sender, RoutedEventArgs e)
```

```
{
   DataEntitiesEmployee = new TitlePresonalEntities();
   ListEmployee.Clear();

   Title title = ComboBoxTitle.SelectedItem as Title;
   ObjectQuery<Employee> employees = DataEntitiesEmployee.Employees;
   var queryEmployee = from employee in employees
                       where employee.TitleID == title.ID
                       orderby employee.Surname
                       select employee;
   foreach (Employee emp in queryEmployee)
   {
      ListEmployee.Add(emp);
   }
   DataGridEmployee.ItemsSource = ListEmployee;
}
```

При запуске данного обработчика выполняется *LINQ-запрос* к базе данных *TitlePersonal* для выборки данных *по* сотрудникам, имеющим заданную должность.

Для обновления выводимого списка сотрудников в *приложение* добавлена *функция* обновления, которая реализуется командой*Refresh*. Данная *команда* добавлена в коллекцию команд страницы *PageEmployee*.

```
<Page.CommandBindings>
....
      <CommandBinding Command="Refresh" Executed="RefreshCommandBinding_Executed" />
...
   </Page.CommandBindings>
```

Вызов команды добавлен в *меню*

```
<MenuItem Header="Обновить" Command="Refresh"/>
```

и *панель инструментов*

```
<Button Name="Refresh" Margin="5,2,5,2"  Command="Refresh"
      ToolTip="обновить данные по сотрудникам">
   <Image Source="Images/Refresh.jpg" ></Image>
</Button>
```

Основное назначение обработчика команды *RefreshCommandBinding_Executed* – обновление списка *ListEmployee*.

```
private void RefreshCommandBinding_Executed(object sender,
ExecutedRoutedEventArgs e)
{
   RewriteEmployee();
   DataGridEmployee.IsReadOnly = false;
   isDirty = false;
   SetUnvisibilityFind();
}
```

Команда "Обновить" обычно используется после поиска данных или ввода данных *по* новому сотруднику.