

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ**  
**Федеральное государственное автономное образовательное учреждение  
высшего образования  
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

**МЕТОДИЧЕСКИЕ УКАЗАНИЯ  
к практическим занятиям  
по дисциплине  
«Объектно-ориентированное программирование»  
для направления подготовки  
09.03.02 Информационные системы и технологии  
Направленность (профиль)  
«Информационные системы и технологии в бизнесе»**

Часть 2

Невинномысск, 2022

## СОДЕРЖАНИЕ

	стр
ВВЕДЕНИЕ	3
Практическое занятие №1 Создание главного окна приложения в среде C#	4
Практическое занятие №2 Создание главного меню приложения	19
Практическое занятие №3 Создание многооконного приложения	24
Практическое занятие №4 Создание пользовательских диалоговых окон	28
Практическое занятие №5 Создание панели инструментов и контекстного меню	39
Практическое занятие №6 Создание строки состояния	44
Практическое занятие №7 Создание элементов управления	49
Практическое занятие №8 Подготовка ADO.NET к работе в приложении	75
Список литературы	90

## ВВЕДЕНИЕ

Работа на практических занятиях по учебной дисциплине «Объектно-ориентированное программирование» предполагает изучение студентами основ объектно-ориентированного программирования.

Изучение дисциплины предполагает формирование компетенции

ПК-4	Способен разработать архитектуру ИС
------	-------------------------------------

Выполнение практических заданий включает:

1. Изучение студентами необходимого теоретического материала по теме лабораторной работы.
2. Постановку задачи в соответствии с темой лабораторной работы и согласование ее с руководителем.
3. Построение алгоритма решения задачи и его документирование в разделе «Краткие теоретические сведения» отчета.
4. Выполнение задания.
5. Подготовку отчета о выполненной работе и его защиту.

Структура отчета по проделанной работе:

1. Тема.
2. Цель занятия.
3. Постановка задачи.
4. Ход выполнения работы.
5. Блок-схема или псевдокод алгоритма решения задачи.
6. Текст программы.
7. Распечатка результатов.
8. Выводы.

# Практическое занятие 1. Создание главного окна приложения в среде C#

Цель занятия: Изучить основные элементы среды разработки Visual Studio Integrated Development Environment ( 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++ - Managed C++, Visual Basic.NET, Java.NET, C#.

В лабораторной работе проводится изучение 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).

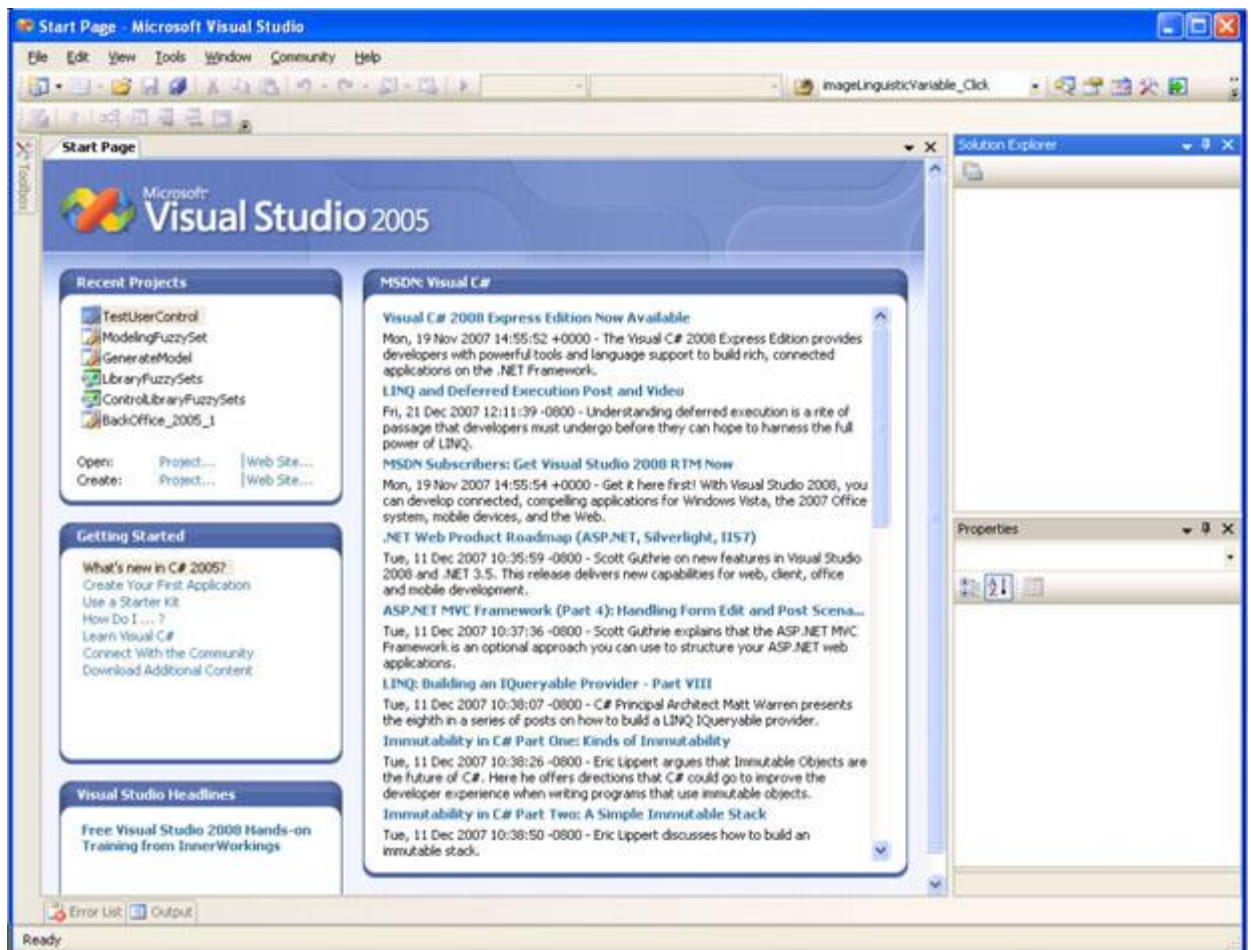
## Обзор среды разработки C#

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.



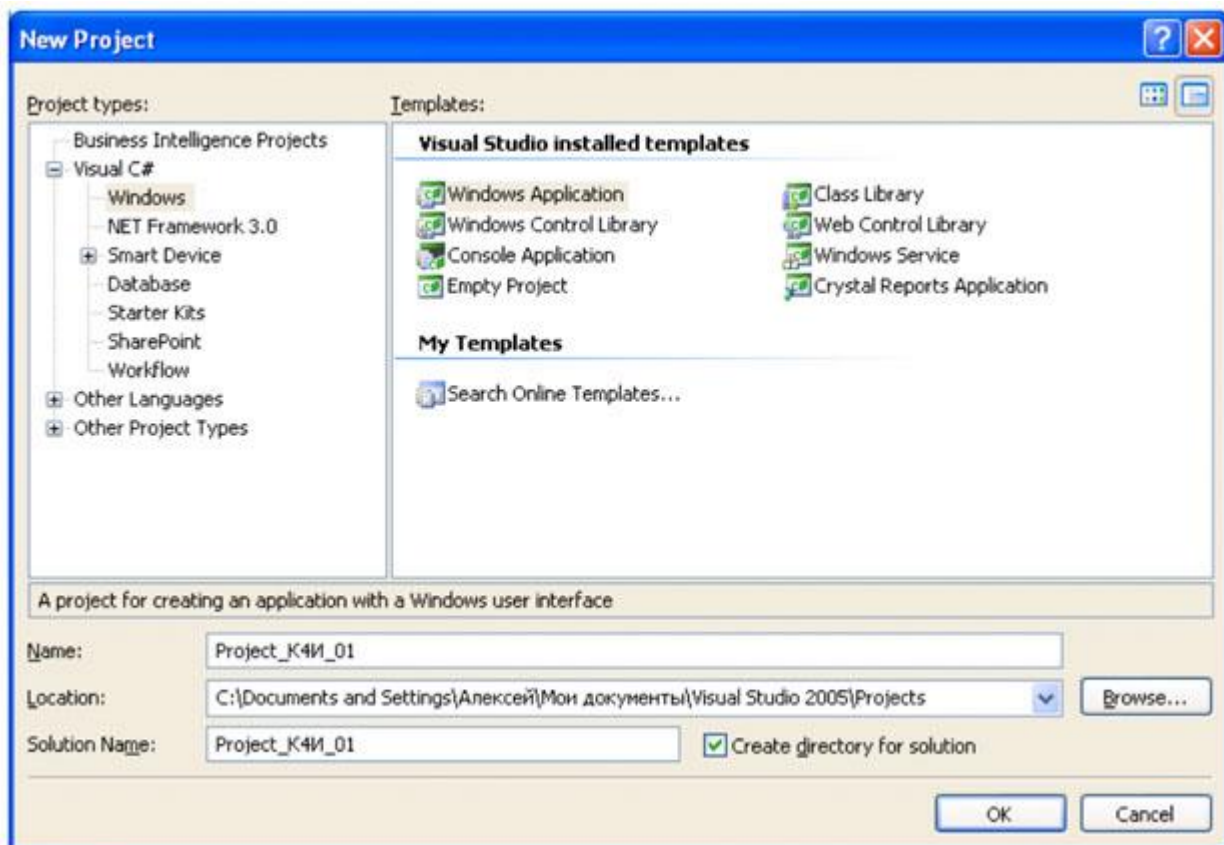
**Рис. 1.1.** Стартовое диалоговое окно IDE

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:

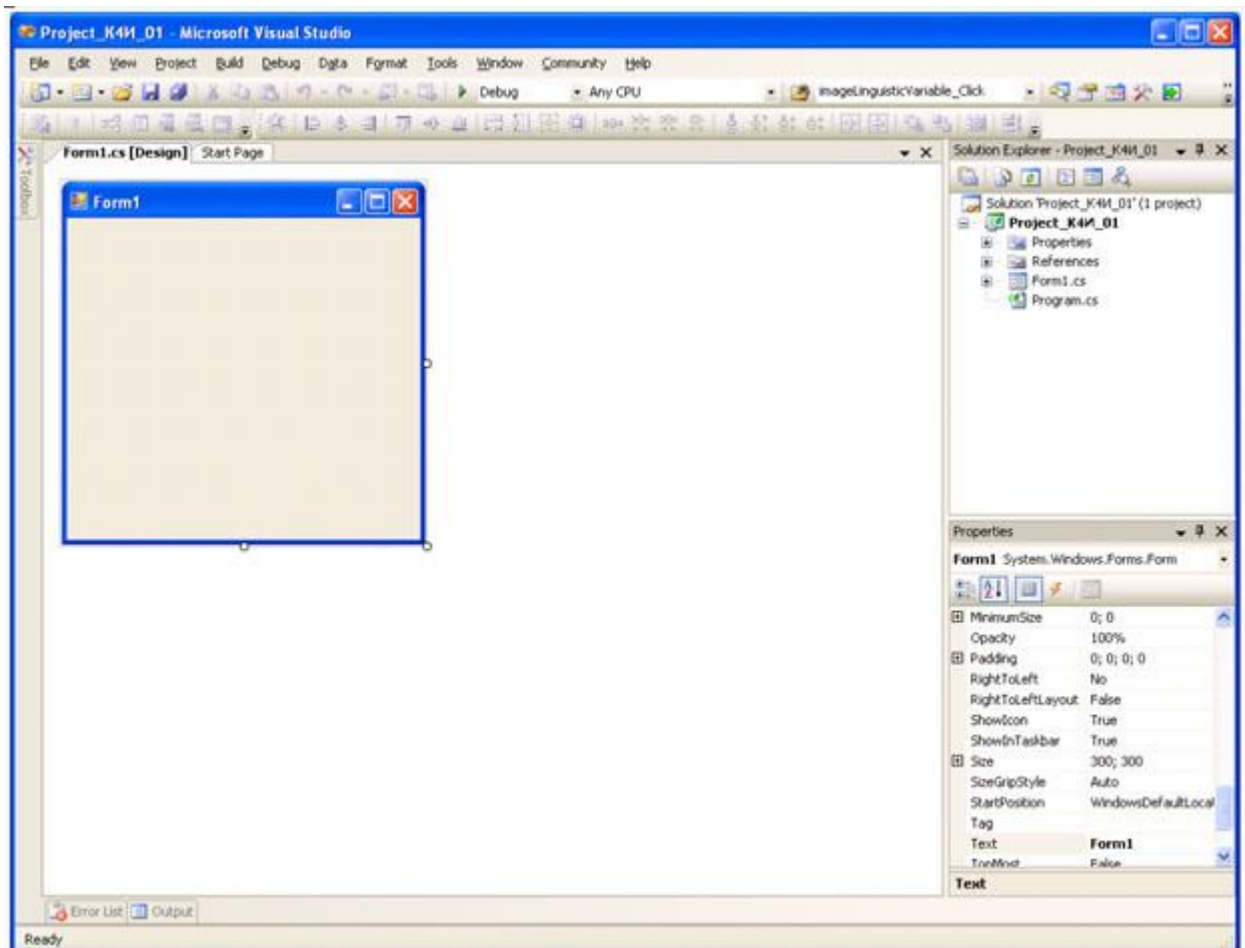
1. Щелкните на ссылке Project (Создать новый проект) метки Create на начальной странице (Start Page) VS.NET. При этом откроется окно создания нового проекта New Project ([рисунок 1.2](#)).



[увеличить изображение](#)

**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_K4И_01` (имя проекта присваивается в соответствии со следующим синтаксисом: `Project_"номер группы"_"номер бригады в группе"` ).
4. Щелкните на кнопке ОК. (Да). Мастер создания нового проекта создаст новый класс `Form1`, производный от `System.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](#)).

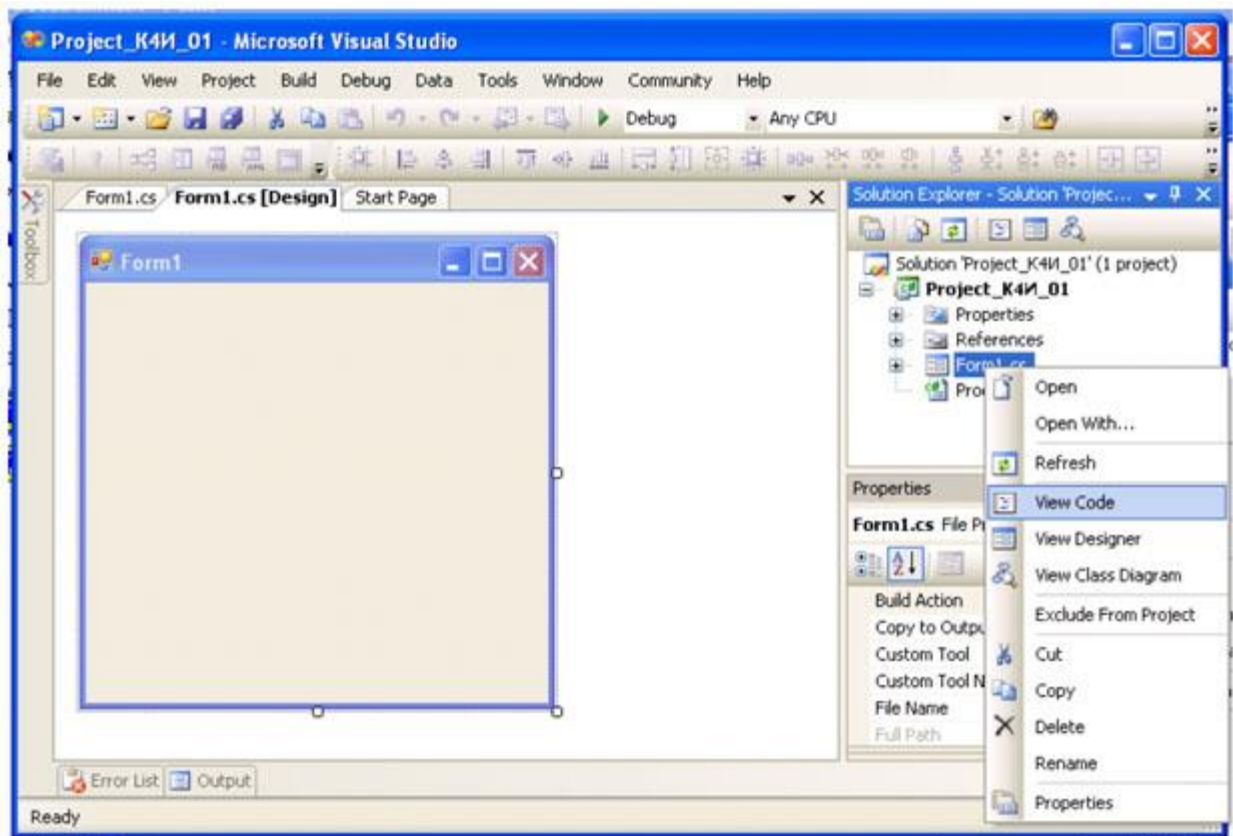


[увеличить изображение](#)

**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 файле `Form1.cs` и в контекстном меню выбрать View Code ([рисунок 1.4](#)).



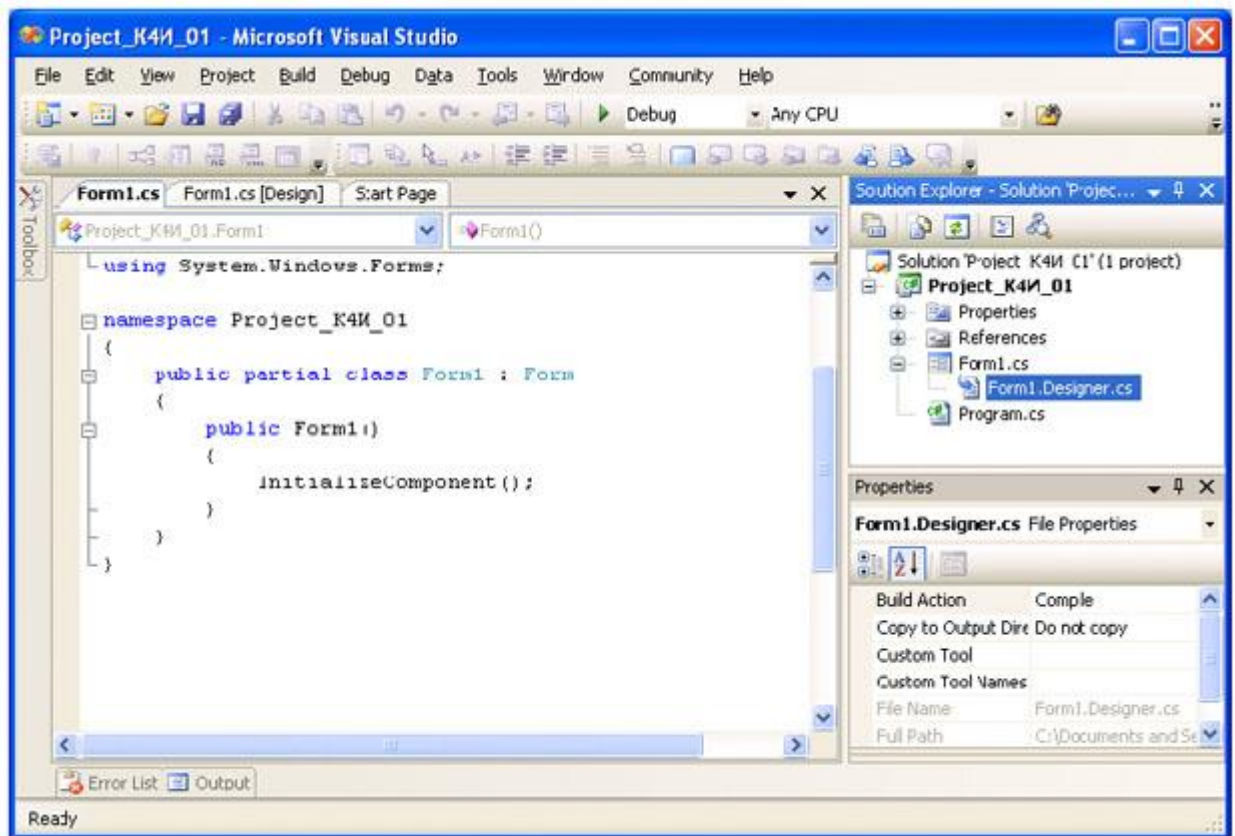
[увеличить изображение](#)

**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_K4И_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](#)).



[увеличить изображение](#)

**Рис. 1.5.** Выбор режима Form.Designer.cs

```

namespace Project_K4M_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_K4И_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 <code>Application</code>
<code>AddMessageFilter( )</code>	Эти методы позволяют приложению перехватывать сообщения <code>RemoveMessageFilter( )</code> и выполнять с этими сообщениями необходимые предварительные действия. Для того чтобы добавить фильтр сообщений, необходимо указать класс, реализующий интерфейс <code>IMessageFilter</code>
<code>DoEvents( )</code>	Обеспечивает способность приложения обрабатывать сообщения из очереди сообщений во время выполнения какой-либо длительной

	операции. Можно сказать, что <code>DoEvents()</code> - это "быстрый и грязный" заменитель нормальной многопоточности
<b>Exit()</b>	Завершает работу приложения
<b>ExitThred()</b>	Прекращает обработку сообщений для текущего потока и закрывает все окна, владельцем которых является этот поток
<b>OLERequired()</b>	Инициализирует библиотеки <b>OLE</b> . Можете считать этот метод эквивалентом <code>.NET</code> для вызываемого вручную метода <code>OleInitialize()</code>
<b>Run()</b>	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
<b>CommonAppDataRegistry</b>	Возвращает параметр системного реестра, который хранит общую для всех пользователей информацию о приложении
<b>CompanyName</b>	Возвращает имя компании
<b>CurrentCulture</b>	Позволяет задать или получить информацию о естественном языке, для работы с которым предназначен текущий поток
<b>CurrentInputLanguage</b>	Позволяет задать или получить информацию о естественном языке для ввода информации, получаемой текущим потоком
<b>ProductName</b>	Для получения имени программного продукта, которое ассоциировано с данным приложением
<b>ProductVersion</b>	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 свойства `Application.CompanyName` без необходимости прибегать к использованию типов, определенных в пространстве имен `System.Reflection`.

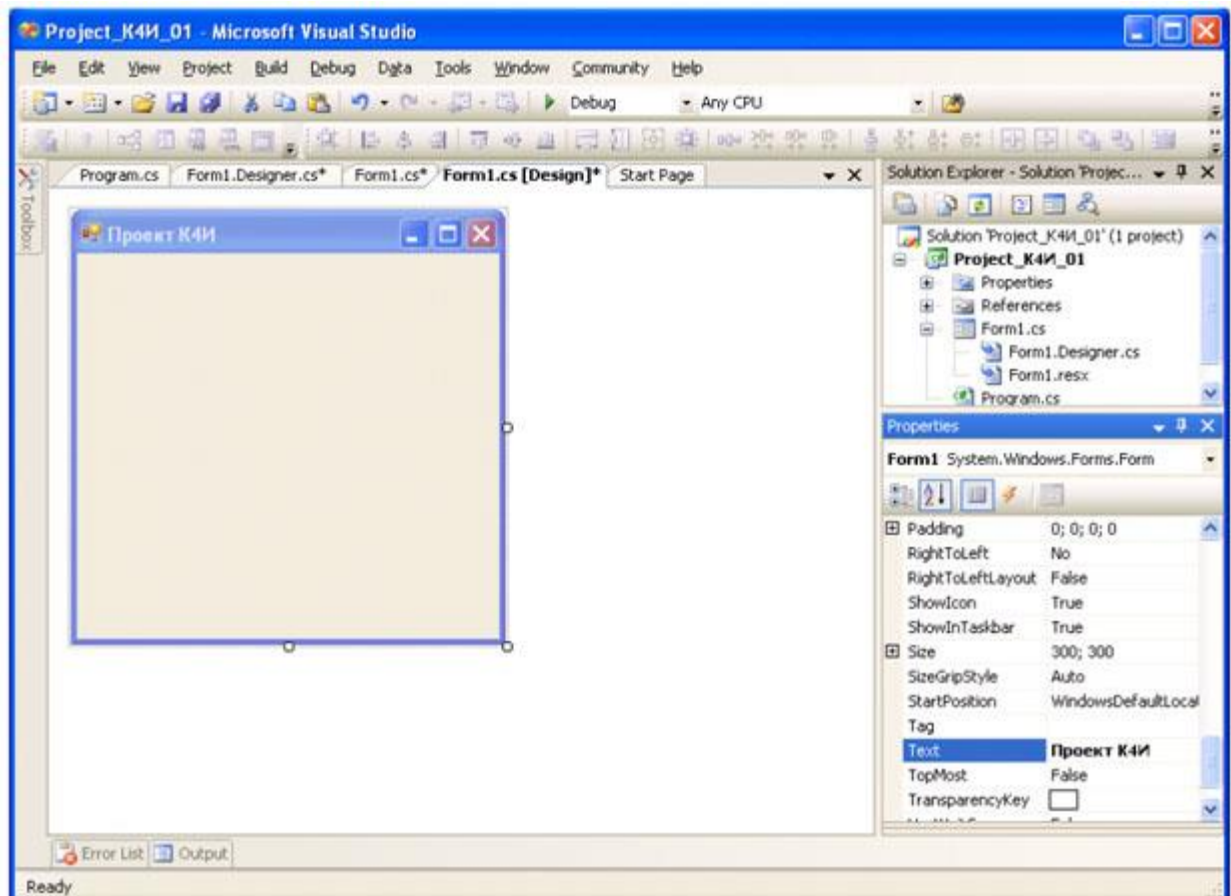
## Проектирование окна приложения

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`.

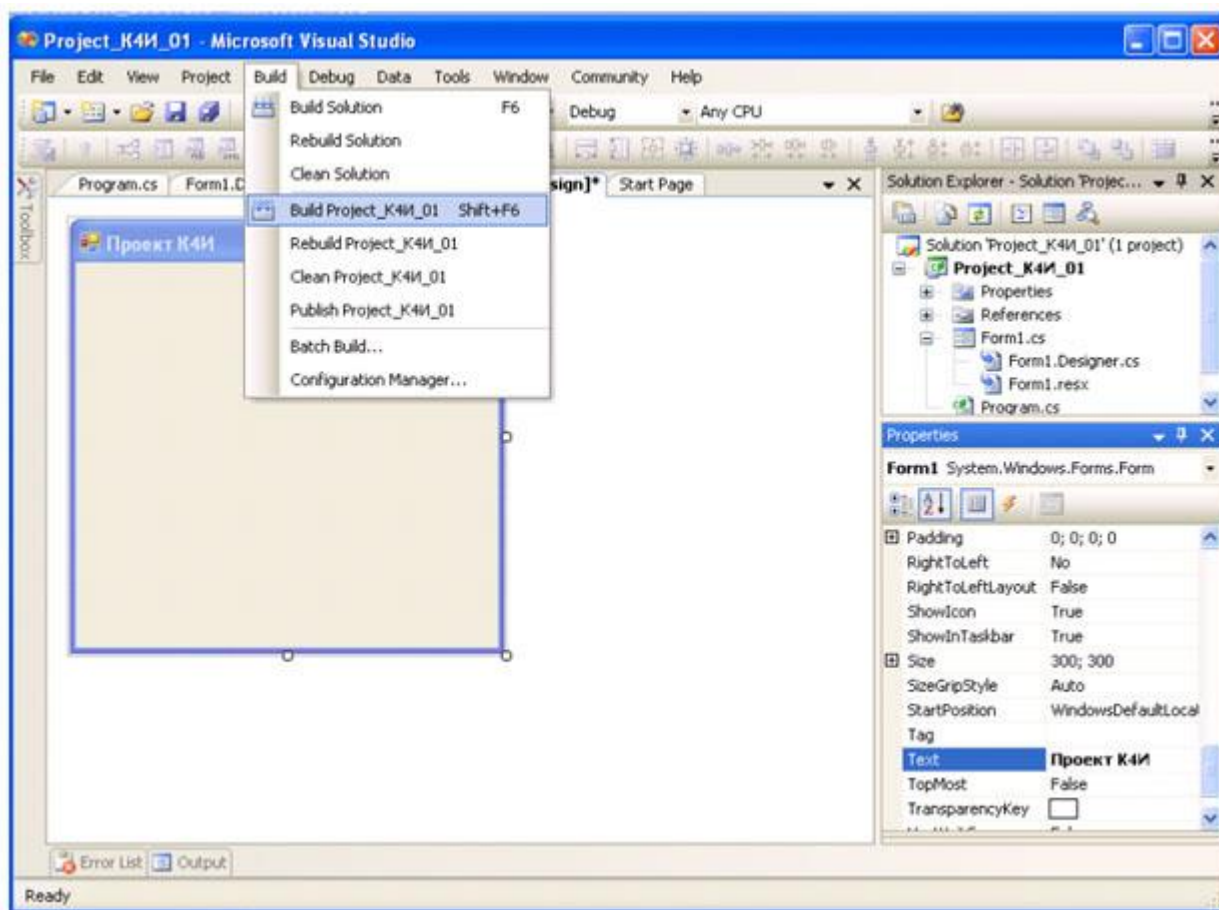
На вкладке `Properties` измените значение в поле `Text` (Заголовок) на Проект К4И. При этом на форме изменится заголовок окна ([рисунк 1.6](#)).



[увеличить изображение](#)

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

Откомпилируйте приложение, выбрав из главного меню команду `Build Project_K4И_01` ([рисунок 1.7](#))



[увеличить изображение](#)

**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.](#)).

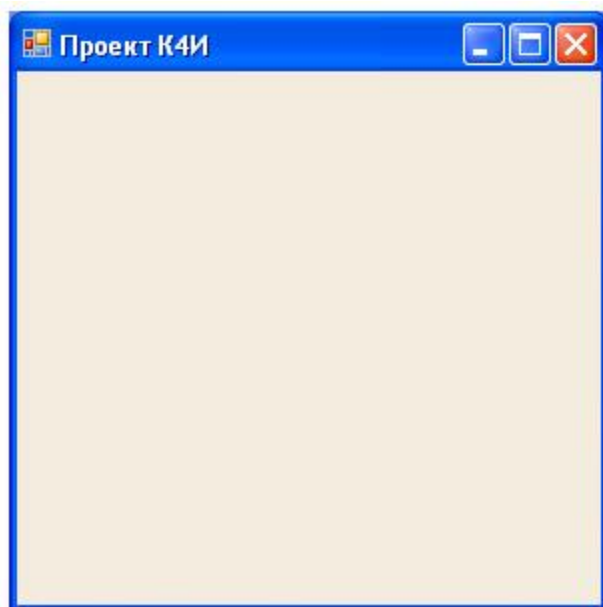
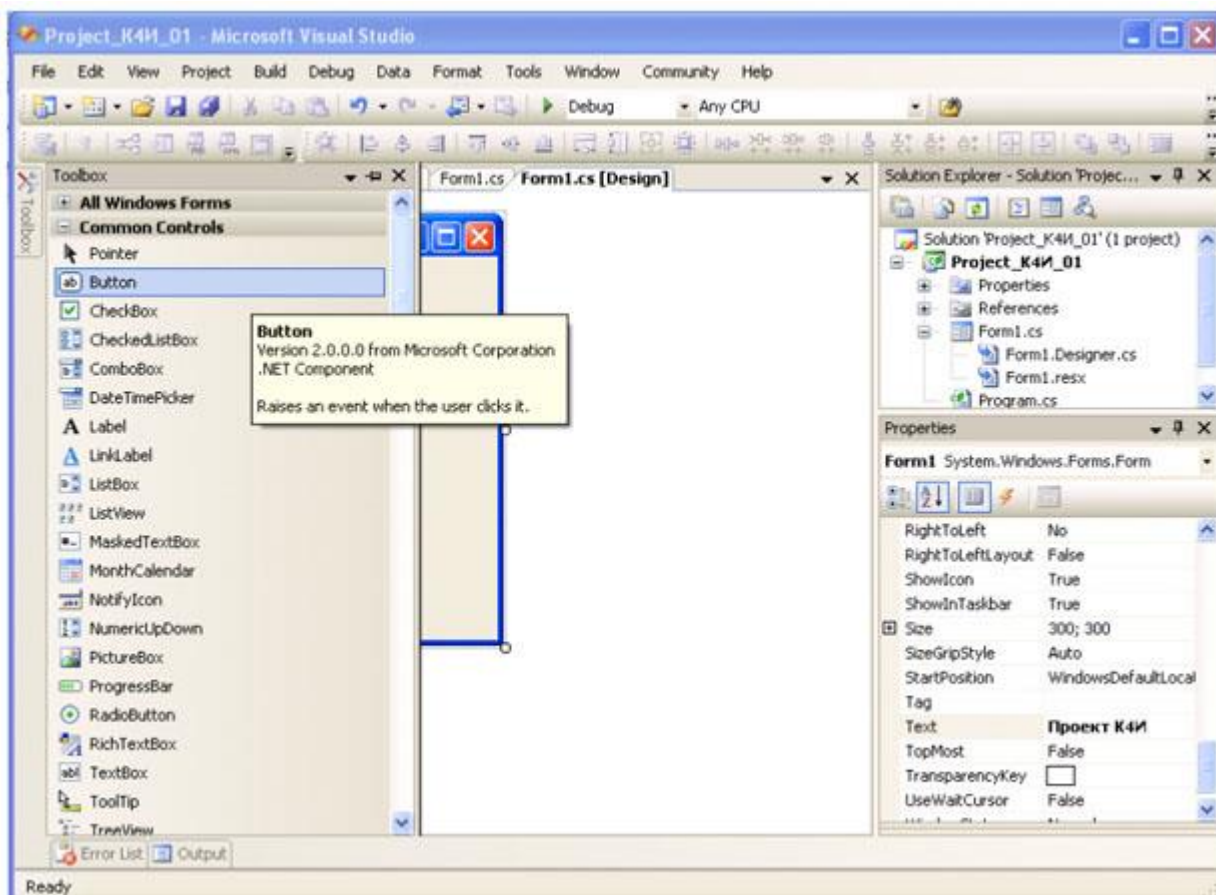


Рис. 1.8. Окно приложения Project\_K4И\_01

Для закрытия окна щелкните мышью на кнопке 

## Добавление нового кода в приложение

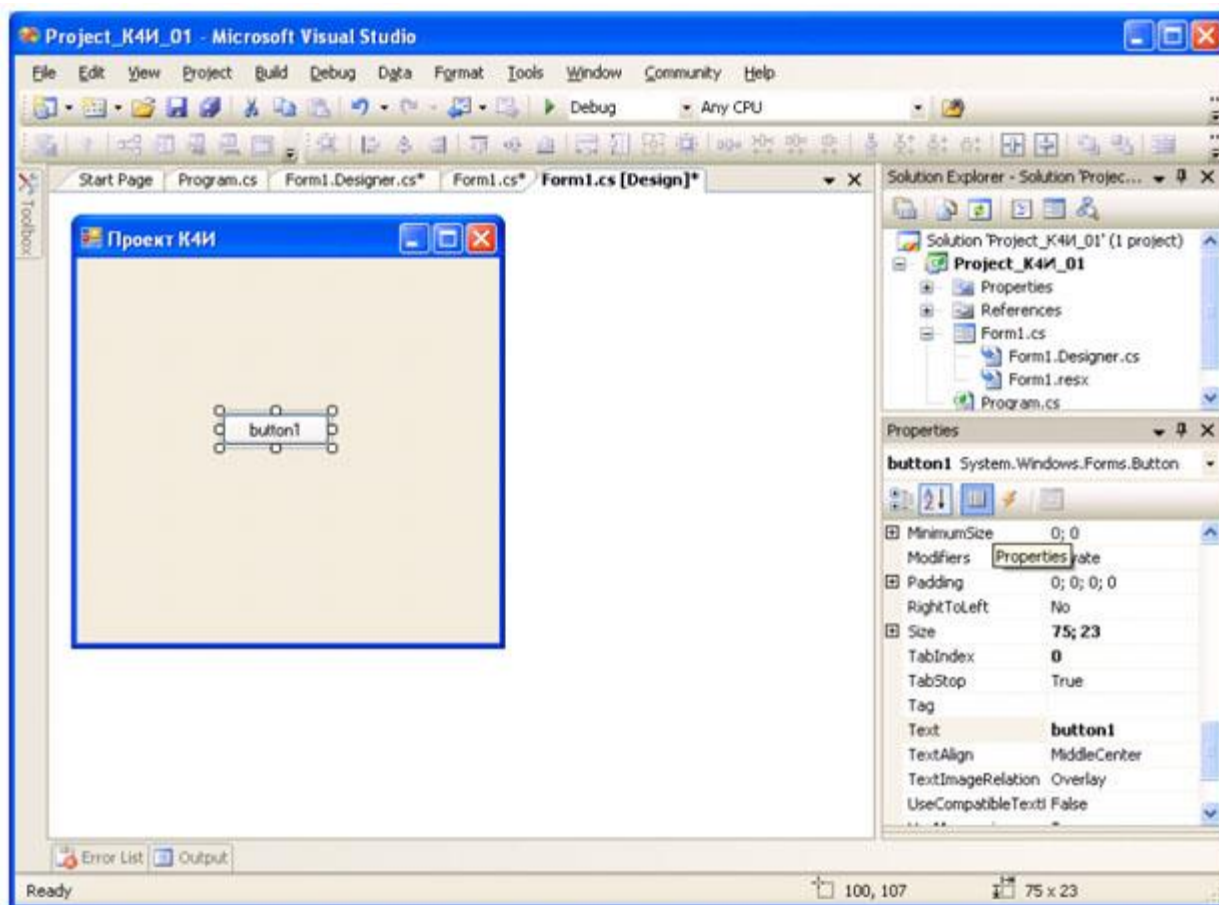
Добавим в главную форму элемент контроля - кнопку. Для этого откроем вкладку ToolBox ([рисунок 1.9](#)) и сначала щелкнем мышью на элементе Button вкладки, а затем щелкнем мышью на форме.



[увеличить изображение](#)

**Рис. 1.9.** Вкладка ToolBox

В результате получим форму с кнопкой ([рисунок 1.10](#)).

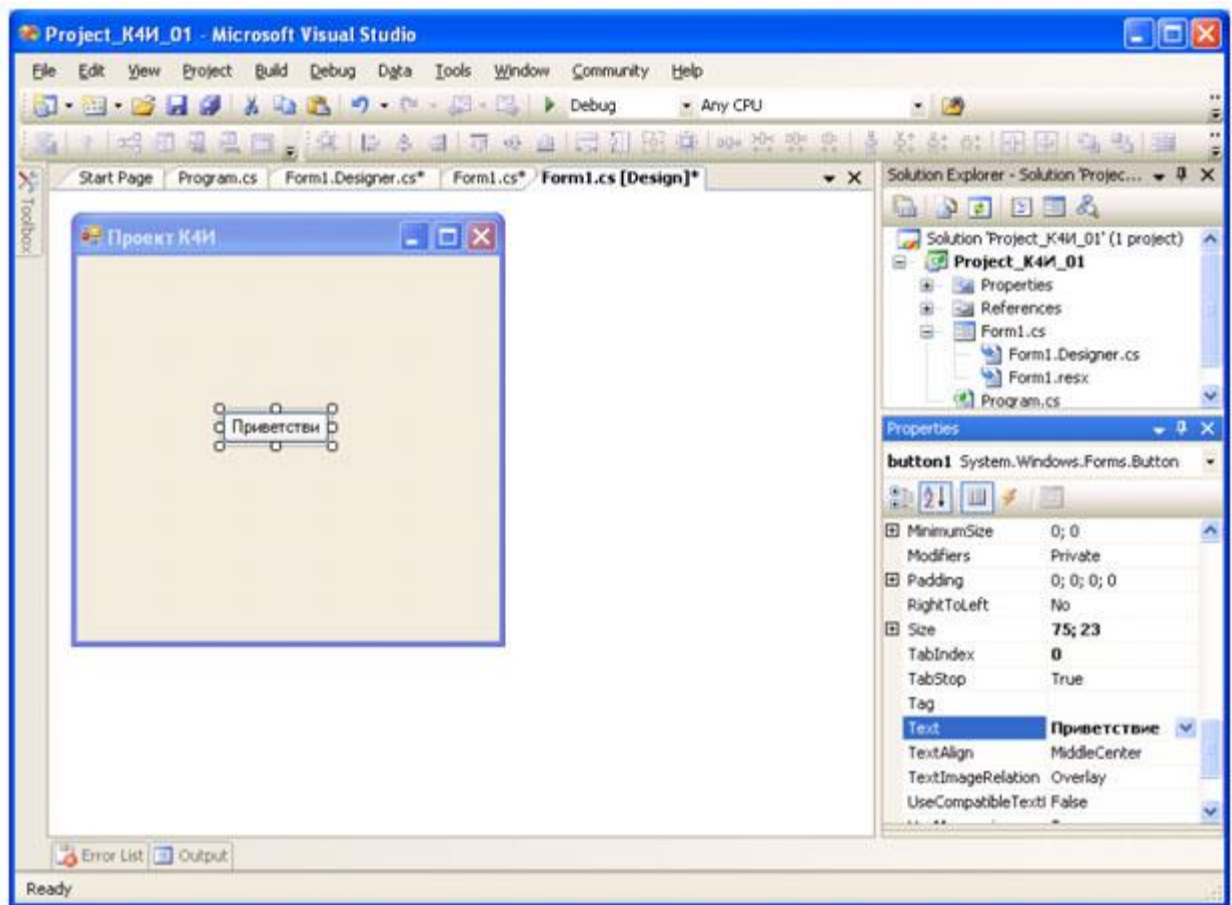


[увеличить изображение](#)

**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](#)).



[увеличить изображение](#)

**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 `Click` для 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](#)).

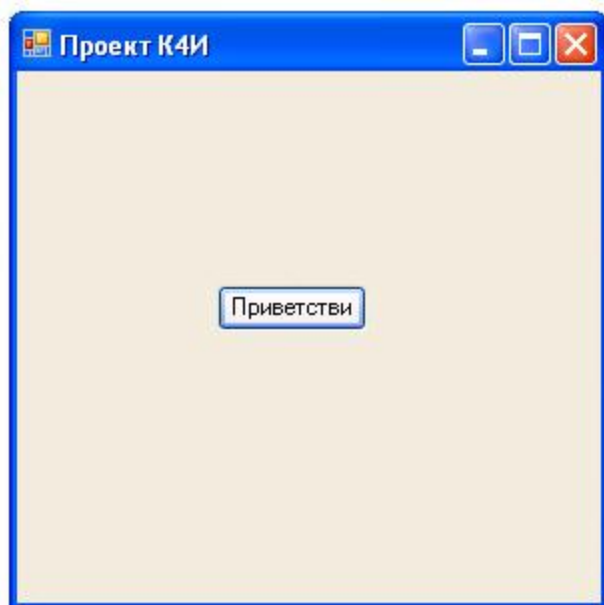


Рис. 1.12. Результат выполнения приложения

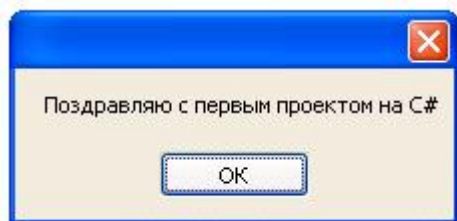


Рис. 1.13. Вывод сообщения

## 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 практическое занятие

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.

## Практическое занятие 2. Создание главного меню приложения

Цель занятия: Изучить основные способы разработки главного *меню* приложения. 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, как `employee_salary`. В других языках часто используют все прописные буквы в названиях констант. Это не рекомендуется в C#, поскольку такие имена трудно читать, лучше применять пascalевский стиль:

```
const int MaximumLength;
```

Еще одна рекомендуемая схема - именование в стиле camel. Именование camel аналогично пascalевскому 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 `public` свойство, 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 называется `ShowConfirmationDialog`, 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.

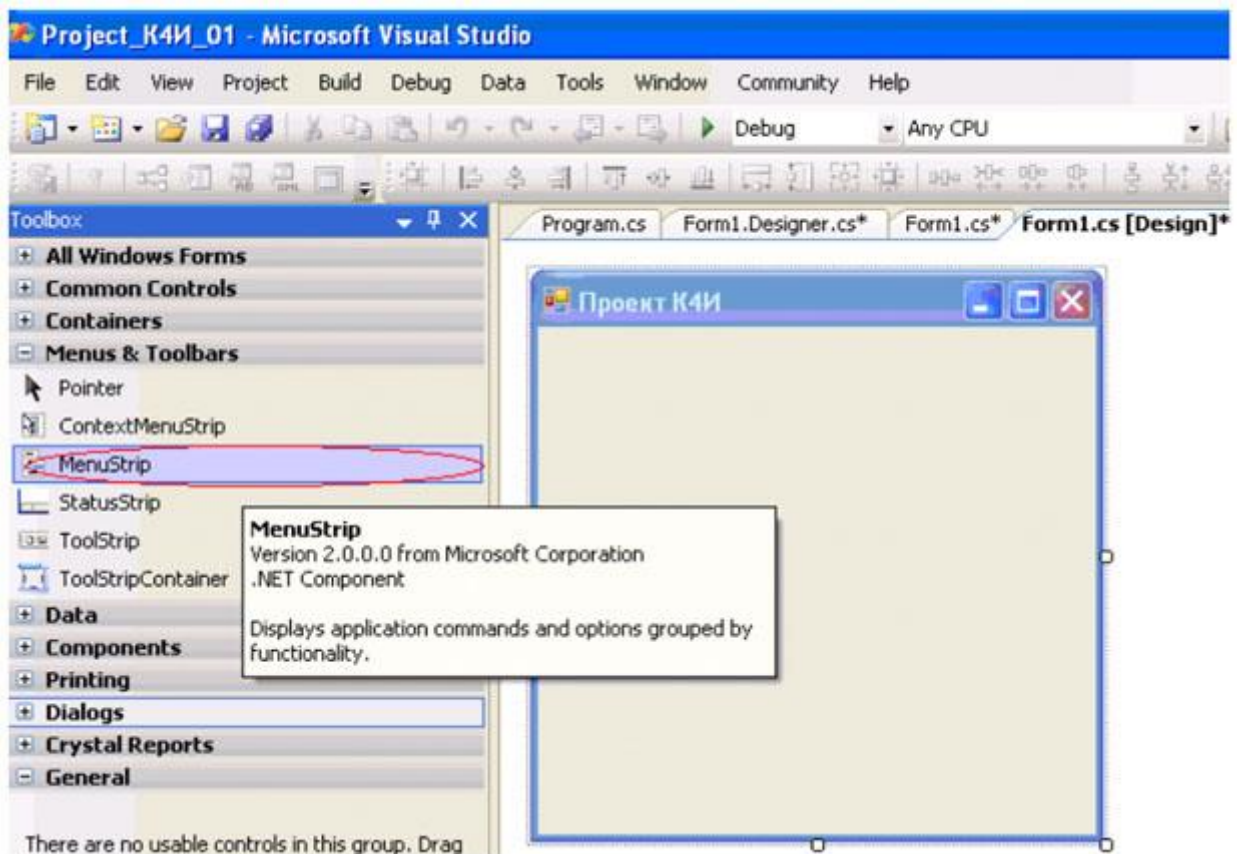
Элемент управления `ToolStrip` представляет собой *контейнер*, используемый для создания структур *меню*, панелей инструментов и строк состояний.

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

*Класс* `ToolStripMenuItem` служит для построения структур *меню*.

Каждый *объект* `ToolStripMenuItem` представляет отдельный *пункт* в системе *меню*.

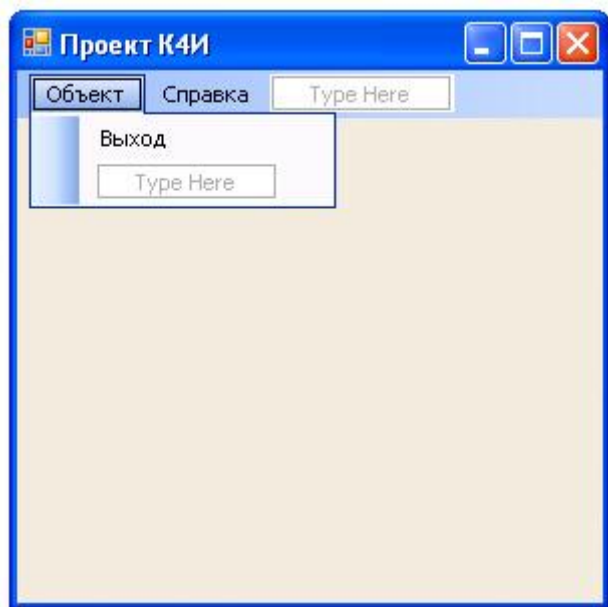
Начнем с создания стандартного ниспадающего *меню*, которое позволит пользователю выйти из приложения, выбрав *пункт* Объект > Выход. Для этого необходимо перетащить элемент управления `MenuStrip` ([РИСУНОК 2.1](#)) на форму в конструкторе.



[увеличить изображение](#)

**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](#).




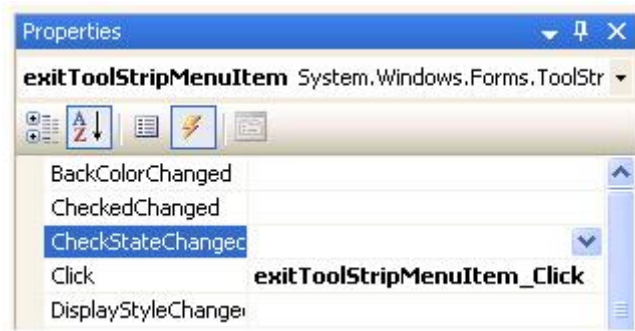
**Рис. 2.2.** Простое меню на форме

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 случае `exitToolStripMenuItem_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 метод `menuItemExit_Click` ([рисунком 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 метод `InitializeComponent` и добавит переменные-члены, представляющие созданные элементы *меню*.

## Задание на практическое занятие

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.

## Практическое занятие 3. Создание многооконного приложения

### Создание дочерней формы

Основа Интерфейса ( MDI ) приложения - MDI родительская форма. Это - форма, которая содержит MDI дочерние окна. Дочерние окна являются "подокнами", с которыми *пользователь* взаимодействует в MDI приложении. Создание MDI родительской формы описано в лабораторной работе 2.

Для определения главного окна ( `Form1` ), как родительской формы в окне Свойств, установите `IsMdiContainer` свойство - `true`. Это определяет форму как MDI *контейнер* для дочерних форм. Для того чтобы родительское окно занимало весь экран необходимо свойству `WindowState` установить *значение* `Maximized`.

Создайте еще одно окно, которое будет дочерним ( `FormEmployee` ). Для этого выберите *пункт меню* Project/Add Windows Form.

Это окно должно вызываться из пункта главного *меню* "Сотрудник". Вставьте код, подобный следующему, чтобы создать новую MDI дочернюю форму, когда *пользователь* щелкает на пункте *меню*, например "Сотрудник" - *имя объекта* - `employeeToolStripMenuItem` (В примере ниже, *указатель* события обращается к событию `Click` для `employeeToolStripMenuItem_Click` ).

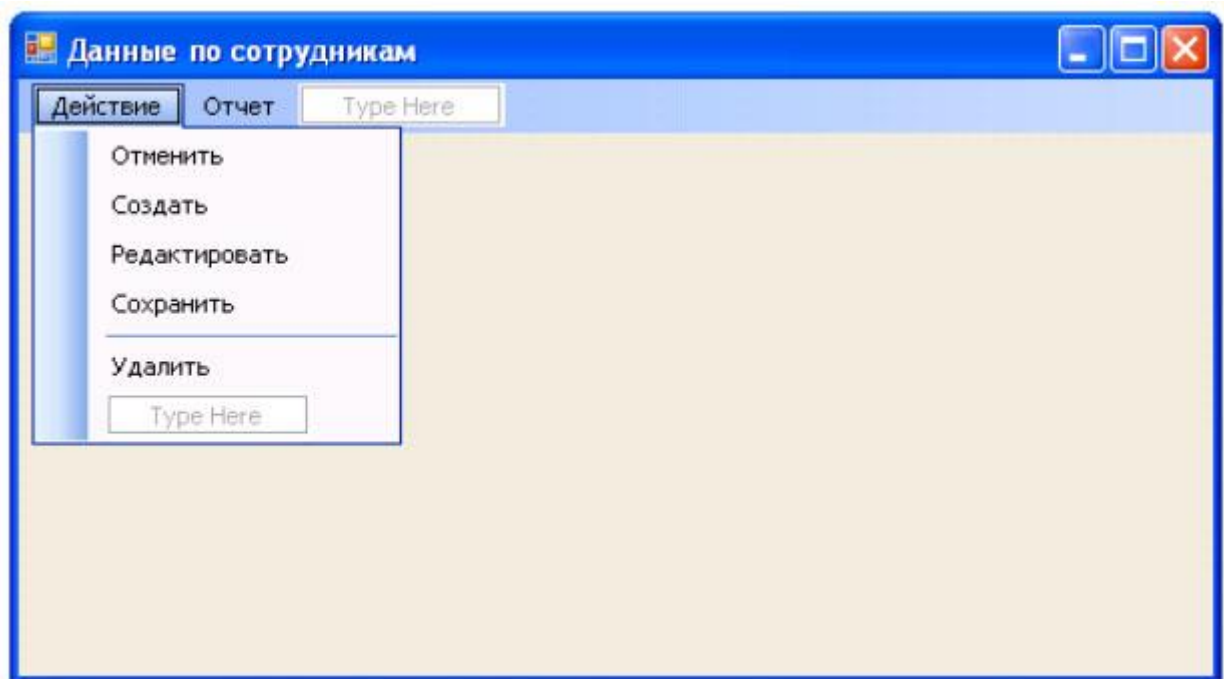
```
private void menuItemEmployee_Click(object sender,
System.EventArgs e)
{ // Создать объект FEmployee класса FormEmployee
FormEmployee FEmployee = new FormEmployee();
    // Установить родительское окно для дочернего
    FEmployee.MdiParent = this;
    // Вывести на экран дочернее окно
    FEmployee.Show();
}
```

Данный обработчик приведет к выводу на экран дочернего окна.

### Создание меню в дочерней форме

Добавьте в дочернее окно *пункт меню* "Действие" ( `actionToolStripMenuItem` ) с подпунктами "Отменить" ( `undoToolStripMenuItem` ), "Создать" ( `createToolStripMenuItem` ), "Редактировать" ( `editToolStripMenuItem` ), "Сохранить" ( `saveToolStripMenuItem` ) и "Удалить" ( `removeToolStripMenuItem` ). Перед пунктом удалить вставьте разделитель ( `Separator - name = toolStripSeparator1` ).

Добавьте в дочернее окно еще один *пункт меню* "Отчет" ( `reportToolStripMenuItem` ) с подпунктами "По сотруднику" ( `reportToolStripMenuItem1` ), "По всем сотрудникам" ( `reportToolStripMenuItem2` ). Дочернее окно будет иметь вид, представленный на [рисунке 3.1](#)

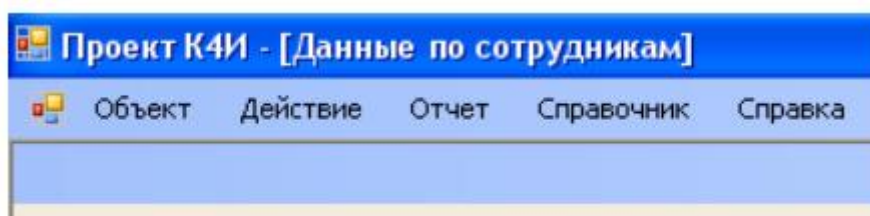


**Рис. 3.1.** Дочернее окно с меню

В главном *меню* родительской формы ( `Form1` ) имеются пункты "Объект", "Справочник" и "Справка". В дочерней форме ( `FormEmployee` ) сформированы пункты *меню* "Действие" и "Отчет". При загрузке дочерней формы *меню* родительской и дочерних форм должны были объединены и составлять следующую последовательность: "Объект", "Действие", "Отчет", "Справочник" и "Справка". *Объединение* пунктов *меню* производится с помощью задания значений свойств `MergeAction` и `MergeIndex` для объектов `ToolStripMenuItem`.

Проверьте, чтобы в *меню* главного окна для объекта `objectToolStripMenuItem` свойство `MergeAction` было установлено `Append`, а `MergeIndex` было равно 0, а для объектов `dictionaryToolStripMenuItem` и `helpToolStripMenuItem` - соответственно 1 и 2. С учетом этого, в окне "Сотрудник" для объектов `actionToolStripMenuItem` (Действие) и "Отчет" ( `reportToolStripMenuItem` ) свойству `MergeAction` необходимо задать *значение* `Insert`, а свойству `MergeIndex` задаем порядковый номер который определяет позицию данного пункта *меню* обновленном главном *меню*, т.е. 1 (после объекта `objectToolStripMenuItem` ).

После компиляции программы, запуска ее на выполнение и вызова пункта *меню* "Сотрудник" экран должен иметь вид, представленный на [рисунке 3.2](#).



**Рис. 3.2.** Дочернее окно с подключенным меню

## Создание обработчиков для меню дочерней формы

Созданные пункты *меню* для дочернего окна должны инициировать выполнение соответствующих функций (Отменить, Создать, Редактировать, Сохранить и Удалить) приложения в отношении объектов конкретного дочернего окна. Для дочернего окна "Данные по сотруднику" эти функции должны выполнять соответственно

отмену редактирования данных *по* сотруднику (*функция* "Отменить"), создавать новые данные *по* сотруднику (*функция* "Создать"), редактировать данные *по* сотруднику (*функция* "Редактировать"), сохранять созданные вновь или отредактированные *функция по* сотруднику (*функция* "Сохранить") и удалять данные *по* сотруднику (*функция* "Удалить").

Описанную функциональность целесообразно реализовать в программе в виде методов класса созданного `FormEmployee`. В приложении необходимо создать следующие методы:

- `Undo` - отменить;
- `New` - создать;
- `Edit` - редактировать;
- `Save` - сохранить;
- `Remove` - удалить.

На начальных этапах проектирования, как правило, неясна реализация каждого метода, поэтому целесообразно их выполнять в виде методов-заглушек, которые только сообщают пользователю о своем вызове, а в дальнейшем необходимо написать реальный код.

Для создания метода `Undo` в коде файла `FormEmployee.cs` добавьте следующий метод:

```
private void Undo( )
{  MessageBox.Show("метод Undo"); }
```

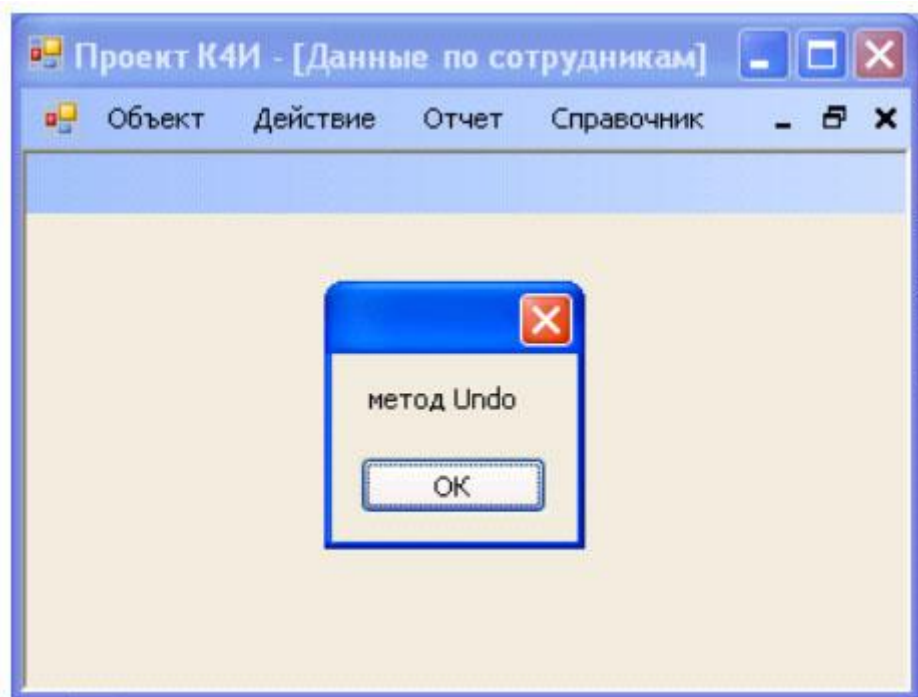
Далее создаем обработчик события вызова пункта *меню* "Отменить". Для этого в дизайнера формы класса `FormEmployee` делаем *двойной щелчок* на пункте *меню* "Отменить". Инструментальная среда VS сгенерирует следующий код:

```
private void undoToolStripMenuItem_Click(object sender, EventArgs e)
{
}
```

В код обработчика `undoToolStripMenuItem_Click` добавим *вызов метода* `Undo`:

```
private void undoToolStripMenuItem_Click(object sender, EventArgs e)
{
    Undo();
}
```

Откомпилируем *приложение* и протестируем *вызов метода* `Undo`. В результате выбора пункта *меню* "Отменить" должно быть выведено *диалоговое окно* с сообщением, приведенным на [рисунке 3.3](#).



**Рис. 3.3.** Дочернее окно с подключенным меню

Аналогичным образом создайте методы-заглушки для функций "Создать", "Редактировать", "Сохранить" и "Удалить".

### **Задание на практическое занятие**

1. Изучить теоретический материал.
2. Создать дочернее окно.
3. В дочернее окно добавить пункты меню.
4. Написать обработчик для вызова из главного меню дочернего окна.
5. Создать коды методов-заглушек для функций приложения.
6. Создать обработчики для вызова пунктов меню.
7. Протестировать работу приложения.

## Практическое занятие 4. Создание пользовательских диалоговых окон

Цель занятия: Изучить основные способы построения диалоговых окон, их параметры и получить практические навыки в разработке

### Основные сведения

Диалоговое окно - это форма, обладающая некоторыми специальными характеристиками. Первая отличительная черта большинства диалоговых окон - то, что их размер изменять нельзя. Кроме того, в диалоговых окнах обычно не используются *элементы управления*, помещаемые в верхнюю часть обычных форм: `ControlBox`, `MinimizeBox` и `MaximizeBox`. Для пользователя *диалоговое окно* в противоположность обычному является практически неизменяемым.

Диалоговые окна бывают модальные и немодальные. Если *приложение* открывает модальное окно, то работа приложения блокируется до тех пор, пока не будет закрыто модальное окно. Немодальные окна могут работать одновременно с породившим их главным окном приложения. Такие окна часто используются для "плавающих" инструментальных панелей, настройки различных параметров приложения, причем отсутствие модальности позволяет использовать в приложении измененные параметры, не закрывая окна настройки этих параметров.

Простейшее модальное *диалоговое окно* можно создать на базе класса `MessageBox`, входящего в библиотеку Microsoft .NET Framework. В лабораторной работе 3 иллюстрировалось применение простейшего модального диалогового окна для вывода сообщения об активизации метода `Undo`. Для отображения диалогового окна использовался метод `Show`, передав ему через *параметр* текст сообщения "`метод Undo`". Прототип использованного метода `Show` следующий:

```
public static DialogResult Show(string message);
```

Когда *пользователь* щелкает кнопку ОК, метод `Show` возвращает *значение*, равное `DialogResult.OK`

Существует множество перегруженных вариантов метода `MessageBox.Show`, позволяющих задать необходимый внешний вид диалоговой панели, а также количество и тип расположенных на ней кнопок.

Прототип наиболее общего варианта метода `MessageBox.Show`, позволяющий реализовать практически все возможности диалогового окна `MessageBox`, приведен ниже

```
public static DialogResult Show
{
    string message, // текст сообщения
    string caption, // заголовок окна
    MessageBoxButtons btns, // кнопки, расположенные в окне
    MessageBoxIcon icon, // значок, расположенный в окне
    MessageBoxDefaultButton defButton, // кнопка по умолчанию
    MessageBoxOptions opt // дополнительные параметры
};
```




*Параметр* `caption` позволяет задать текст заголовка диалогового окна `MessageBox`. С помощью параметра `btns` можно указать, какие кнопки необходимо расположить в окне диалогового окна. Этот *параметр* задается константами из перечисления `MessageBoxButtons` ([таблица 4.1](#))

Таблица 4.1. Перечисление MessageBoxButtons

Константа	Кнопки, отображаемые в окне MessageBox
<code>OK</code>	OK
<code>OKCancel</code>	OK, Cancel
<code>YesNo</code>	Yes, No
<code>YesNoCancel</code>	Yes, No, Cancel
<code>RetryCancel</code>	Retry, Cancel
<code>AbortRetryIgnore</code>	Abort, Retry, Ignore

Параметр `icon` метода `MessageBox.Show` позволяет выбрать один из нескольких значков для расположения в левой части диалогового окна. Он задается в виде константы перечисления `MessageBoxIcon` ([таблица 4.2](#)).

Таблица 4.2. Перечисление MessageBoxIcon

Константа	Значок
<code>Asterisk, Information</code>	
<code>Error, Stop</code>	
<code>Exclamation, Warning</code>	

Question	
None	Значок не отображается

Параметр `defButton` метода `MessageBox.Show` предназначен для выбора кнопки, которая получит фокус сразу после отображения диалогового окна. Этот параметр должен иметь одно из значений перечисления `MessageBoxDefaultButton` ([таблица 4.3](#)).

Таблица 4.3. Перечисление <code>MessageBoxDefaultButton</code>	
Константа	Номер кнопки, получающей фокус ввода по умолчанию
<code>Button 1</code>	1
<code>Button 2</code>	2
<code>Button 3</code>	3

Если в диалоговом окне отображаются кнопки Yes, No и Cancel, то они будут пронумерованы последовательно: кнопка Yes получит номер 1 (константа `Button1`), кнопка No - номер 2 (константа `Button2`), а кнопка Cancel - номер 3 (константа `Button3`).

Параметр `opt` метода `MessageBox.Show` позволяет задать дополнительные параметры. Эти параметры должны иметь значения из перечисления `MessageBoxOptions` ([таблица 4.4](#)).

Таблица 4.4. Перечисление <code>MessageBoxOptions</code>	
Константа	Описание
<code>DefaultDesktopOnly</code>	Окно с сообщением отображается только на рабочем столе, выбранном по умолчанию
<code>RightAlign</code>	Текст сообщения выравнивается по правому краю диалогового окна

<code>RtlReading</code>	Текст отображается справа налево
<code>ServiceNotification</code>	Окно отображается на активном рабочем столе, даже если к системе не подключился ни один пользователь. Данный параметр предназначен для приложений, работающих как сервисы ОС Microsoft Windows

Если в окне диалогового окна `MessageBox` имеется несколько кнопок, то для того, что бы определить, какую кнопку щелкнул *пользователь*, программа должна проанализировать *значение*, возвращенное методом `MessageBox.Show`.

Метод `MessageBox.Show` может вернуть одно из нескольких значений перечисления `DialogResult` ([таблица 4.5](#)).

Таблица 4.5. Перечисление DialogResult	
Константа	Кнопка, при щелчке которой возвращается эта константа
<code>Abort</code>	Abort
<code>Cancel</code>	Cancel
<code>Ignore</code>	Ignore
<code>No</code>	No
<code>None</code>	Модальное диалоговое окно продолжает работать
<code>OK</code>	OK
<code>Retry</code>	Retry
<code>Yes</code>	Yes

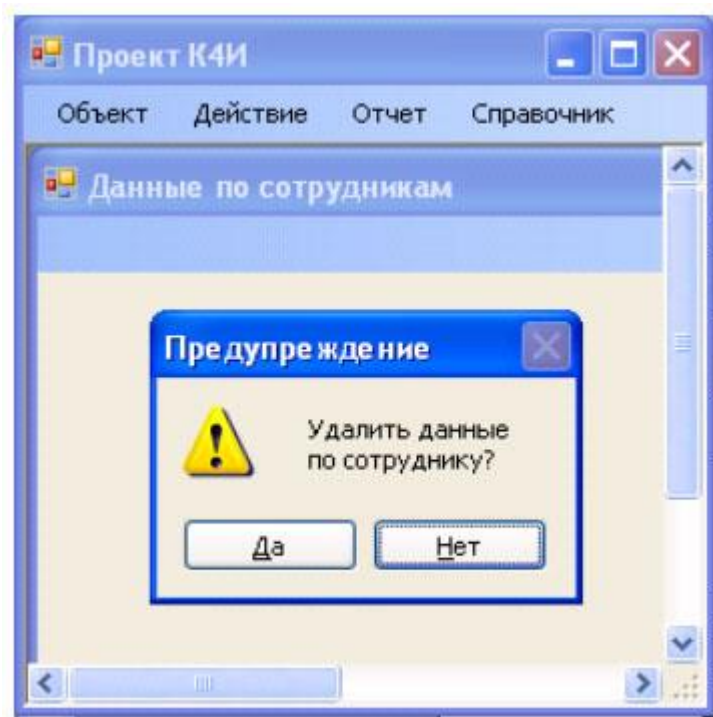
Изменим метод `Remove`, добавив в него предупреждение перед удалением данных *по* сотруднику. Текст кода метода `Remove` должен иметь следующий вид:

```

private void Remove()
{
    DialogResult result = MessageBox.Show(" Удалить данные \n по сотруднику?
",
    "Предупреждение", MessageBoxButtons.YesNo, MessageBoxIcon.Warning,
    MessageBoxDefaultButton.Button2);
    switch (result)
    {
        case DialogResult.Yes:
            {
                //выполнить действия по удалению данных по сотруднику
                MessageBox.Show("Удаление данных");
                break;
            }
        case DialogResult.No:
            {
                //отмена удаления данных по сотруднику
                MessageBox.Show("Отмена удаления данных");
                break;
            }
    }
}

```

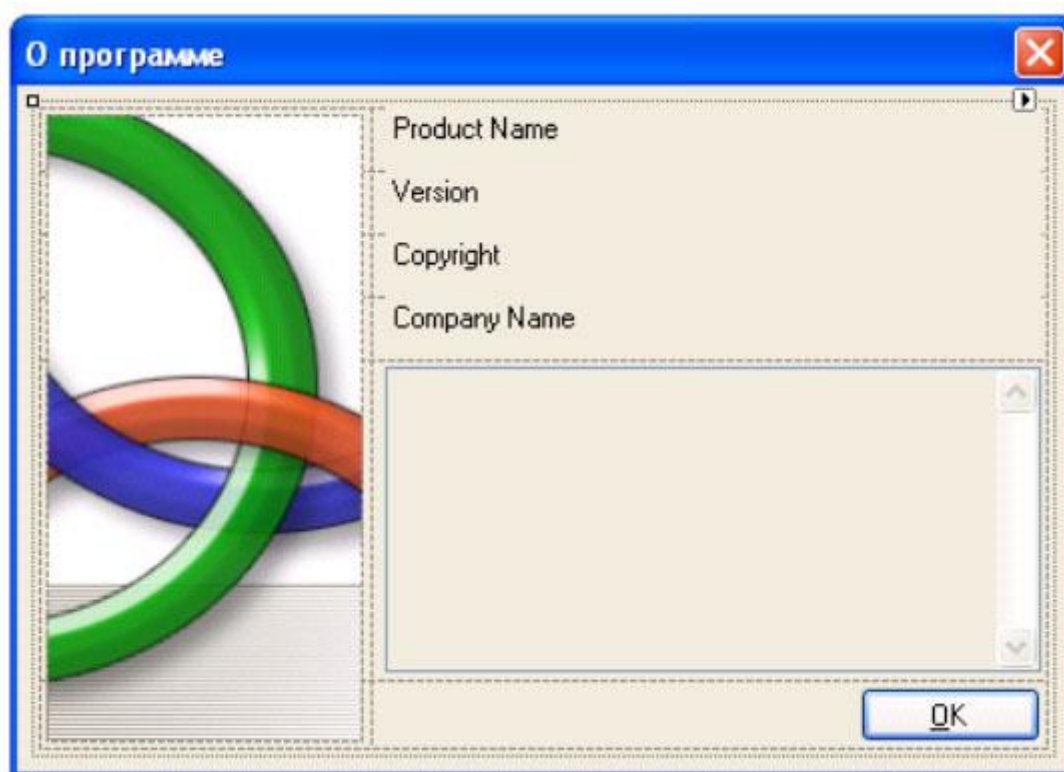
В результате исполнения кода приложения и выбора пункта *меню* "Удалить" будет выводиться предупреждение, приведенное на [рисунке 4.1](#) .



**Рис. 4.1.** Модальное диалоговое окно предупреждения

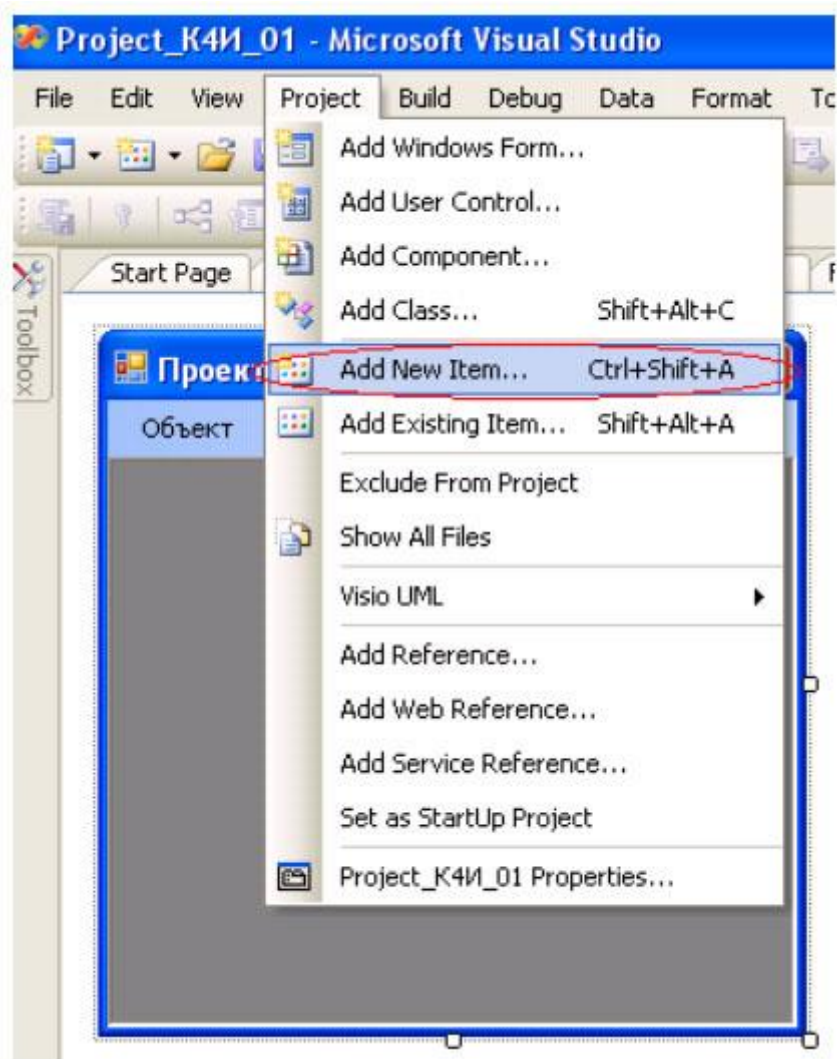
*Диалоговое окно* можно создать не только на основе класса `MessageBox`, но и с использованием *Windows - формы*.

Создадим новую форму FormAbout для вывода справочной информации о разрабатываемом приложении, которое должно иметь вид представленный на [рисунке 4.2](#) .

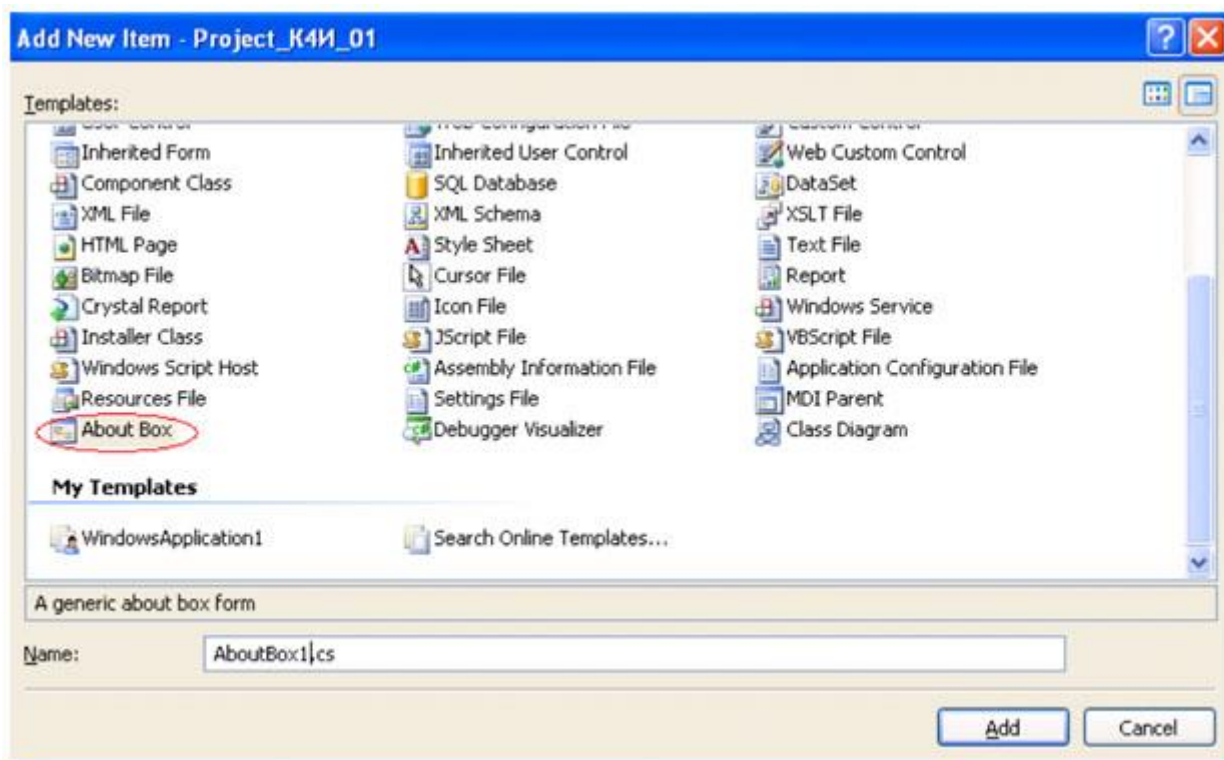


**Рис. 4.2.** Общий вид Windows - формы FormAbout

Для этого добавим в проект новый *компонент* ([рисунком 4.3](#)), выбрав из списка `AboutBox` ( [рис. 4.4](#)).



**Рис. 4.3.** Выбор режима добавления нового компонента в проект



[увеличить изображение](#)

**Рис. 4.4.** Добавление нового компонента в проект

Для класса `AboutBox` можно задать *ЛОГОТИП* и дополнительную информацию. По умолчанию данный *класс* берет дополнительную информацию из метаданных сборки. Проверьте это.

Мы введем собственную информацию. Для этого изменим фрагмент кода конструктора класса `AboutBox1` следующим образом.

```
public AboutBox1()
{
    InitializeComponent();
    this.Text = String.Format("О программе {0}", AssemblyTitle);
    this.labelProductName.Text = AssemblyProduct;
    this.labelVersion.Text = String.Format("Version {0}", AssemblyVersion);
    this.labelCopyright.Text = "@РГЭУ, 2008";
    this.labelCompanyName.Text = "Долженко А.И.";
    this.textBoxDescription.Text = "Дисциплина Современные технологии
программирования. Студенческий проект";
}
```

Для открытия пользовательского модального диалогового окна используется метод `ShowDialog`. В лабораторной работе *диалоговое окно* должно открываться при щелчке пользователем на пункте в *меню* "Справка/О программе". Код для открытия диалогового окна может выглядеть следующим образом:

```
// Открываем модальное диалоговое окно
private void aboutToolStripMenuItem_Click(object sender, EventArgs e)
{
```

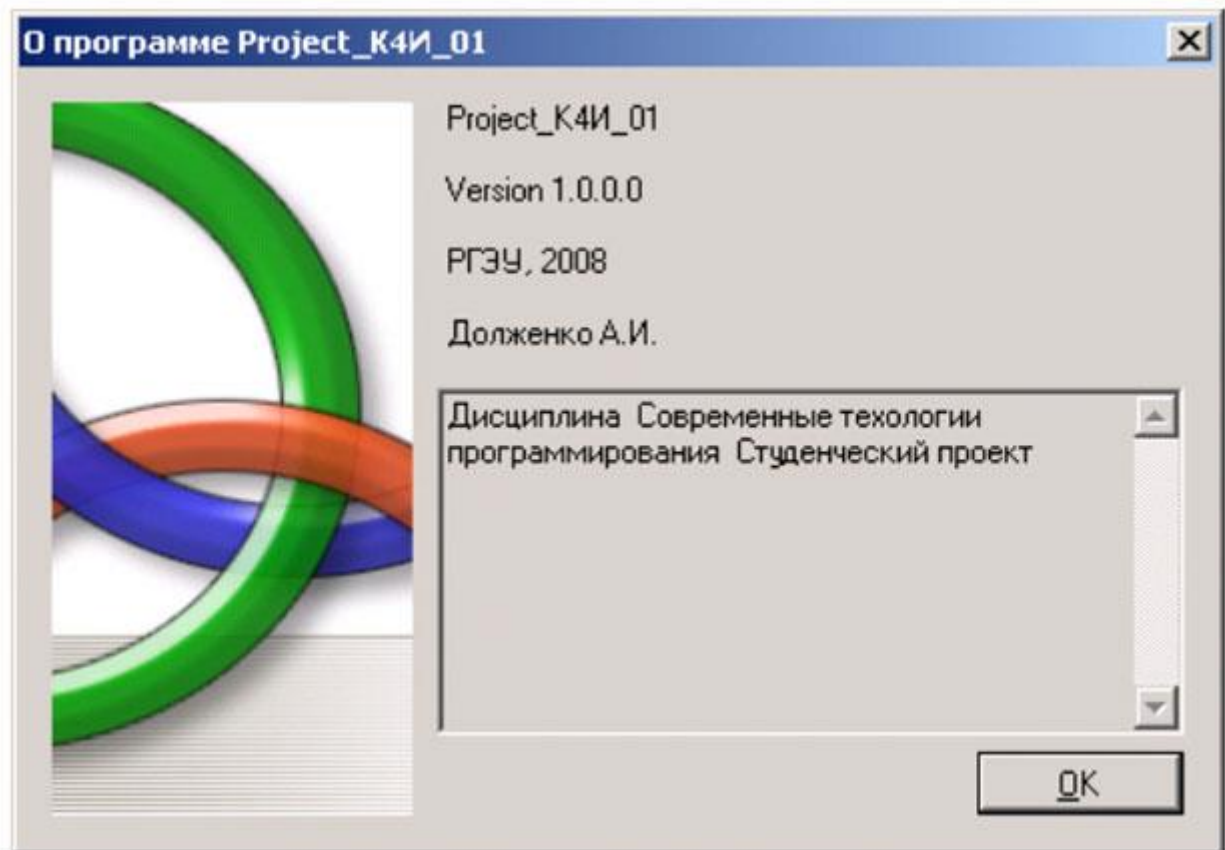
```

AboutBox1 aboutBox = new AboutBox1();
aboutBox.ShowDialog(this);
}

```

*Модальность формы* определяет именно метод `ShowDialog`: при использовании кода ход выполнения программы будет приостановлен вплоть до того момента, пока метод `ShowDialog` не вернет соответствующее значение. Для пользователя это значит, что ему придется закрыть *диалоговое окно*, прежде чем он сможет выполнить какие-либо *операции* на главной форме.

После компиляции и загрузки приложения, вызвав пункт меню "Справка/О программе" на дисплеи будет выведено *диалоговое окно*, приведенное на [рисунке 4.5](#).



[увеличить изображение](#)

**Рис. 4.5.** Вызов модального окна

При нажатии на кнопку `OK` *диалоговое окно* будет автоматически закрыто и в ходе дальнейшего выполнения программы можно выяснить значение свойства `DialogResult`.

### Задание на практическое занятие

1. Изучить теоретический материал.
2. Создать модальное диалоговое окно с помощью класса `MessageBox`.
3. Создать пользовательское модальное диалоговое окно для пункта меню "О программе".
4. Написать обработчики для вызова модальных окон.
5. Протестировать работу приложения.

## Практическое занятие 5. Создание панели инструментов и контекстного меню

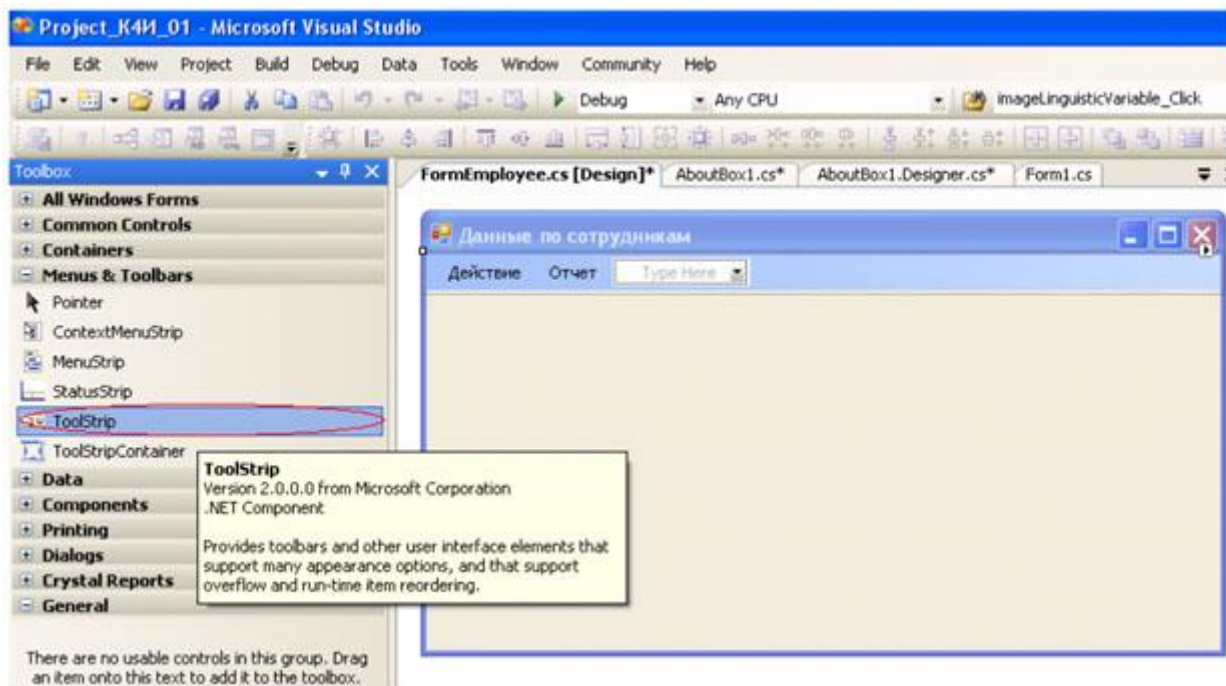
Цель занятия: Изучить основные способы построения панели инструментов и контекстного *меню*, их параметры и получить практические навыки в разработке.

В приложении для повышения качества интерфейса пользователя целесообразно предусматривать различные способы активизации функций системы. При выполнении лабораторной работы 3 для приложения было создано *меню*, которое позволяет активизировать функции "Отменить", "Создать", "Редактировать", "Сохранить" и "Удалить". Данные функции могут быть активизированы с помощью кнопок панели инструментов и контекстного *меню*.

### Разработка панели инструментов

Элемент управления ToolStrip используется непосредственно для построения панелей инструментов. Данный элемент использует набор элементов управления, происходящих от класса `ToolStripItem`.

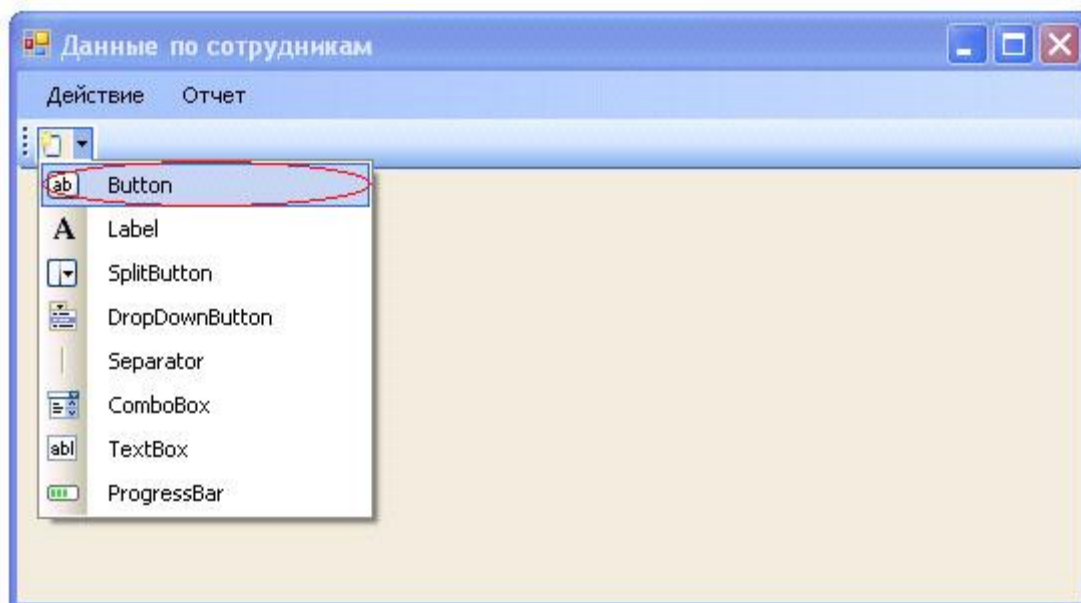
В Visual Studio.NET предусмотрены средства, которые позволяют добавить *панель инструментов* при помощи графических средств. Для этого необходимо открыть панель Toolbox и добавить элемент управления ToolStrip ([рисунок 5.1](#)) на разрабатываемую форму FormEmployee.



[увеличить изображение](#)

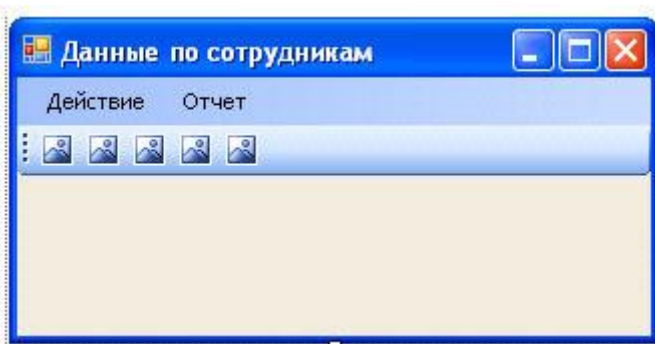
**Рис. 5.1.** Окно свойств панели инструментов

В выпадающем *меню* элемента управления ToolStrip на форме FormEmployee необходимо выбрать элемент управления button - кнопка ([рисунок 5.2](#)). При этом в панели инструментов добавится кнопка.




**Рис. 5.2.** Окно свойств панели инструментов

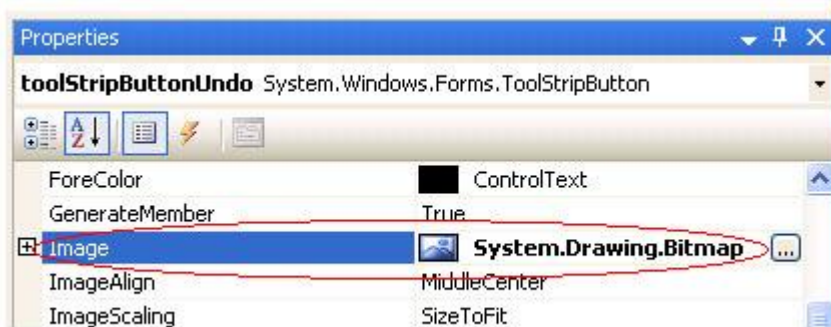
Добавьте на *панель инструментов* кнопки с именами `toolStripButtonUndo`, `toolStripButtonNew`, `toolStripButtonEdit`, `toolStripButtonSave`, `toolStripButtonRemove`. В результате должна быть сформирована *панель инструментов* с кнопками ([рисунок 5.3](#)).



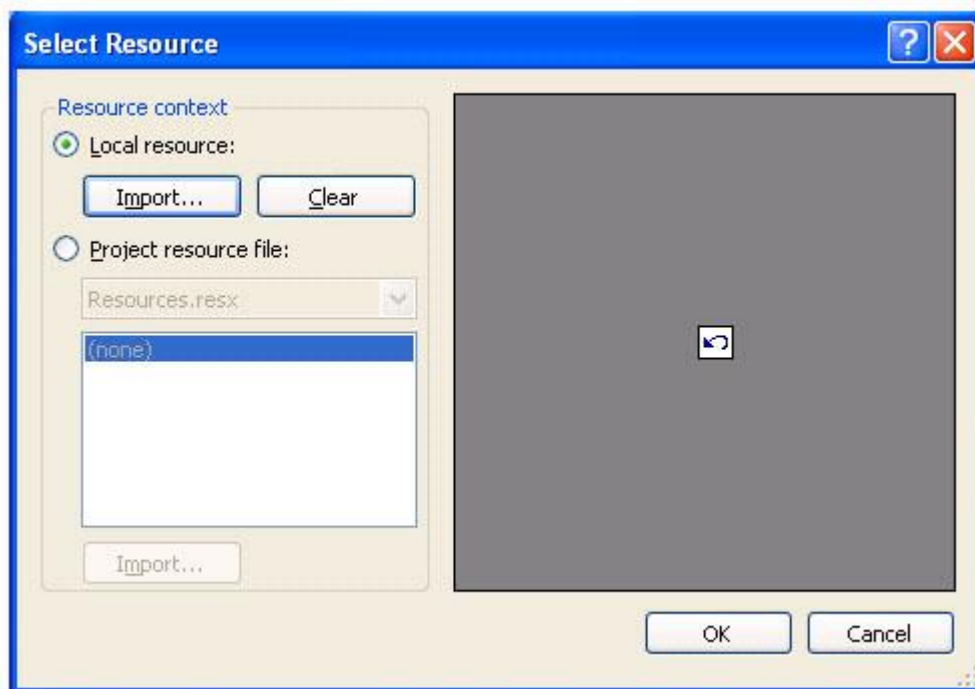
**Рис. 5.3.** Форма FormEmployee с панелью инструментов

Для кнопок панели инструментов сформируем графическое *представление*. Это можно сделать путем задания свойства `Image` соответствующей кнопке ([рисунок 5.4](#)).

При открытии коллекции свойства `Image` соответствующей кнопки, нажатии кнопки  открывается окно мастера выбора графического ресурса ([рисунок 5.5](#)).

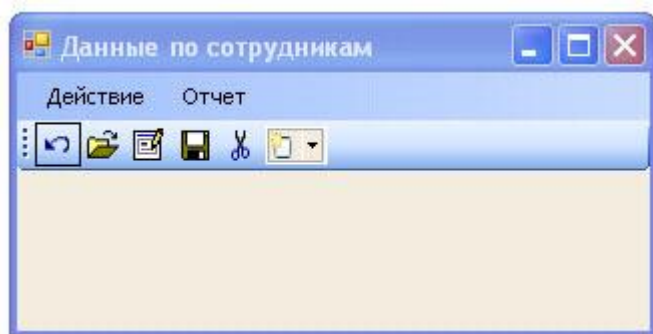


**Рис. 5.4.** Свойство Image для кнопки панели инструментов



**Рис. 5.5.** Добавление изображения в ImageList

С помощью кнопки Import в локальный ресурс добавляют ссылки на необходимые графические файлы, для формирования изображения кнопок. Результаты формирования графического представления кнопок *панель инструментов* приведены на [рисунке 5.6](#). Графические файлы расположены в папке *Visual Studio2005\VS2005ImageLibrary\bitmaps\commands\16color* (для лабораторной работы графические файлы можно найти в папке *Лабораторные работы*).

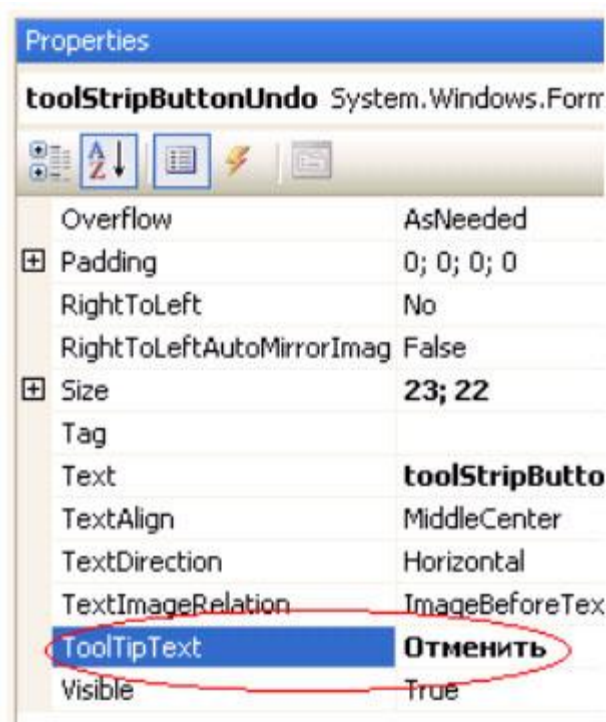


**Рис. 5.6.** Форма с панелью инструментов

Каждая кнопка панели инструментов, которая является объектом класса `toolStripButton`, может содержать текст, или изображение, или и то и другое.

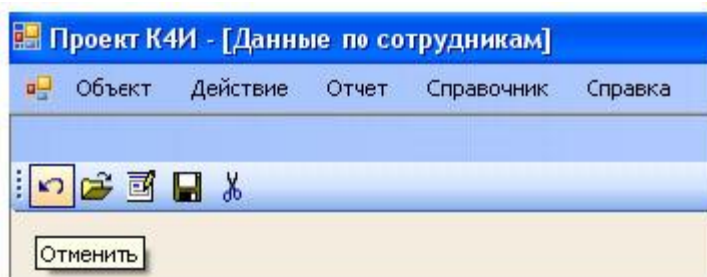
Созданная *панель инструментов* содержит пять кнопок. По функциональности каждой из этих кнопок будут соответствовать пункты *меню* "Отменить", "Создать", "Редактировать", "Сохранить" и "Удалить".

Для удобства пользователя целесообразно снабдить кнопки панели инструментов всплывающими подсказками при фокусировке курсора на данной кнопке. Это можно сделать, если свойству `ToolTipText` класса `toolStripButton` задать *значение* текстовой строки с содержанием подсказки. На [рисунке 5.7](#) для кнопки "Отменить" (`toolStripButtonUndo`) строка подсказки `ToolTipText` соответствует строке "Отменить".



**Рис. 5.7.** Формирование подсказки для кнопки

На [рисунке 5.8](#) показан *вывод* подсказки при фокусировке курсора на кнопке  панели инструментов.



**Рис. 5.8.** Вывод подсказки для кнопки

Для распознавания реакции приложения при нажатии кнопок панели инструментов необходимо создать обработчик события для кнопок. При двойном нажатии на кнопку панели инструментов генерируется обработчик, в который нужно добавить *вызов метода* `Undo`. В этом случае обработчик нажатия кнопки панели инструментов будет иметь следующий вид:

```
private void toolStripButtonUndo_Click(object sender, EventArgs e)
{
    Undo();
}
```

## Контекстное меню

Класс `ContextMenuStrip` применяется для показа контекстного меню, или меню, отображаемого по нажатию правой кнопки мыши. Для создания объекта класса `ContextMenuStrip` необходимо открыть панель Toolbox и добавить элемент управления `contextMenuStrip` на форму `FormEmployee` (рисунок 5.9).

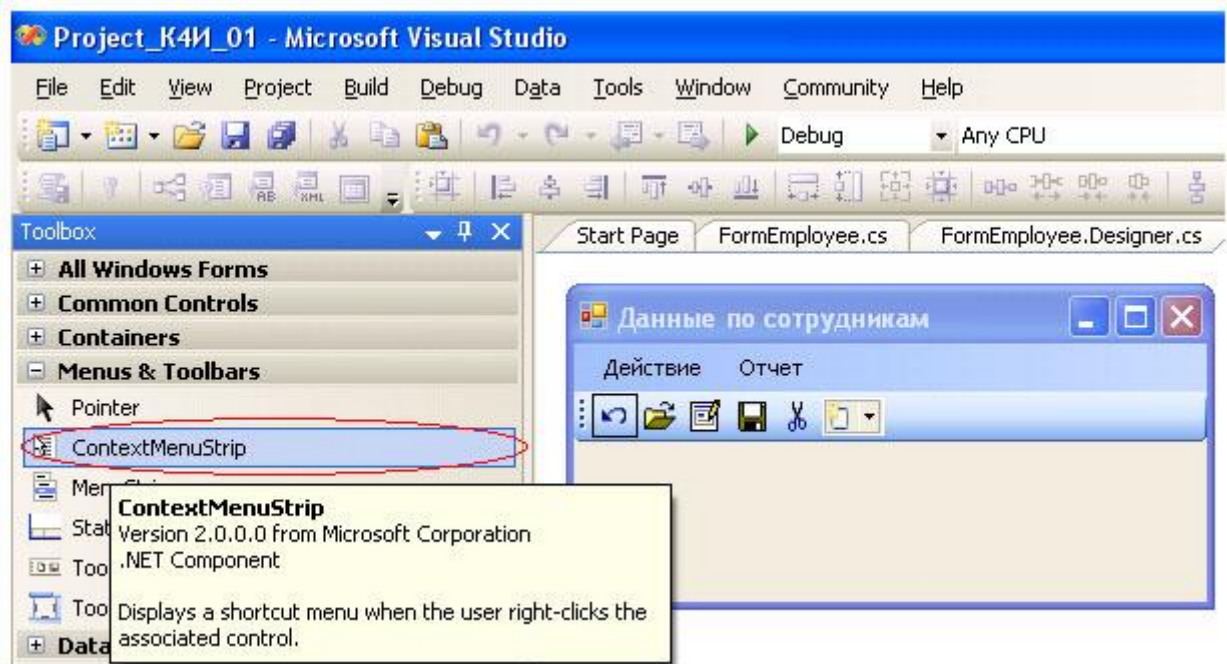


Рис. 5.9. Формирование на форме контекстного меню

В результате получаем форму `FormEmployee` с контекстным меню (рисунок 5.10)

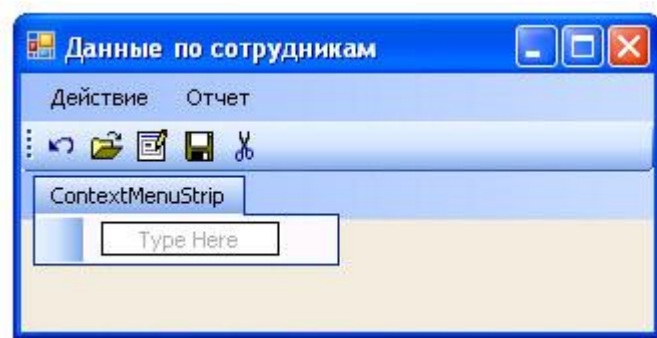
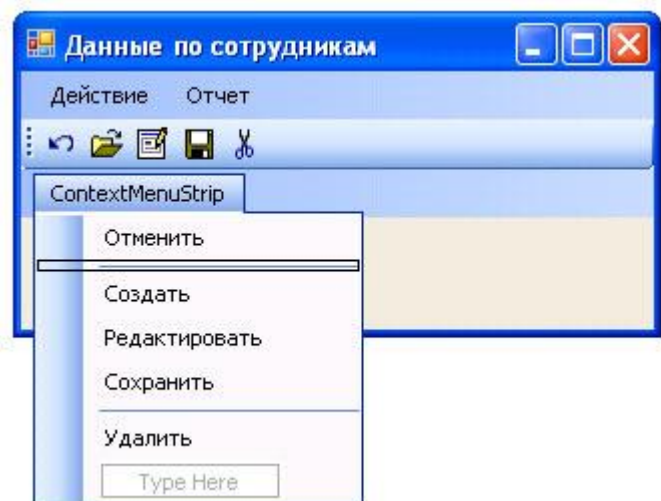


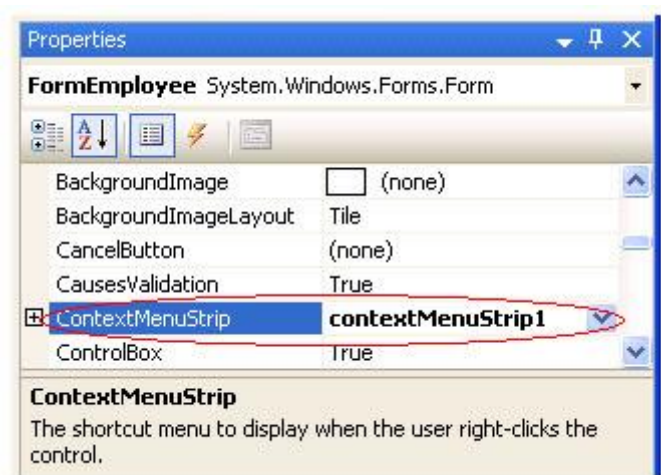
Рис. 5.10. Форма с контекстным меню

Формирование пунктов контекстного меню производится аналогично формированию пунктов главного меню (смотри лабораторную работу 2). Сформированное контекстное меню приведено на рисунке 5.11.



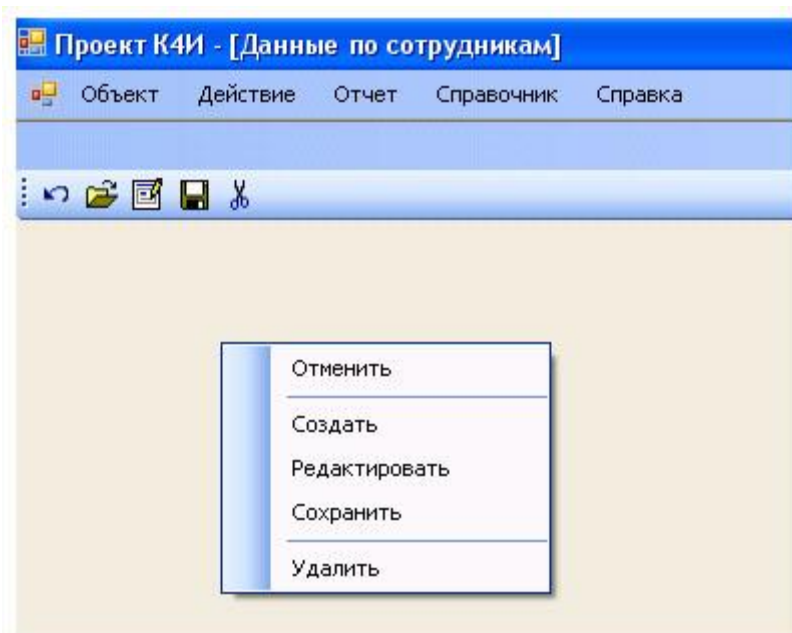
**Рис. 5.11.** Вид контекстного меню

После формирования пунктов контекстного *меню* необходимо его подключить к форме FormEmployee. Для этого на вкладке Свойства ( Properties ) формы FormEmployee строке, соответствующей свойству `ContextMenuStrip` нужно установить *значение* созданного объекта `contextMenuStrip1` ( [рис. 5.12](#) )



**Рис. 5.12.** Подключение контекстного меню к форме

После компиляции проекта и запуска приложения на выполнение можно проверить режим активизации контекстного *меню*. Для этого необходимо выбрать из главного *меню* пункт "Сотрудник" и на появившейся форме в любом месте щелкнуть правой кнопкой мыши. Результат всплытия на форме контекстного *меню* показан на [рисунке 5.13](#).



**Рис. 5.13.** Активизация контекстного меню

Привязка пунктов контекстного *меню* к конкретным функциям осуществляется путем создания кода обработчика событий для каждого пункта *меню*. Для формирования обработчика необходимо перейти в окно дизайнера формы FormEmployee, выделить на форме *класс* ContextMenuStrip и сделать *двойной щелчок* на соответствующем пункте *меню*, например "Отменить". В сгенерированном обработчике необходимо добавить *вызов метода*, для функции "Отменить" - метод Undo ( ). Листинг обработчика метода приведен ниже.

```
private void undo1ToolStripMenuItem_Click(object sender, EventArgs e)
{
    Undo ();
}
```

### **Задание на практическое занятие**

1. Изучить теоретический материал.
2. Создать панель инструментов.
3. Создать контекстное меню.
4. Написать обработчики для панели инструментов и контекстного меню.
5. Протестировать работу приложения

## Практическое занятие 6. Создание строки состояния

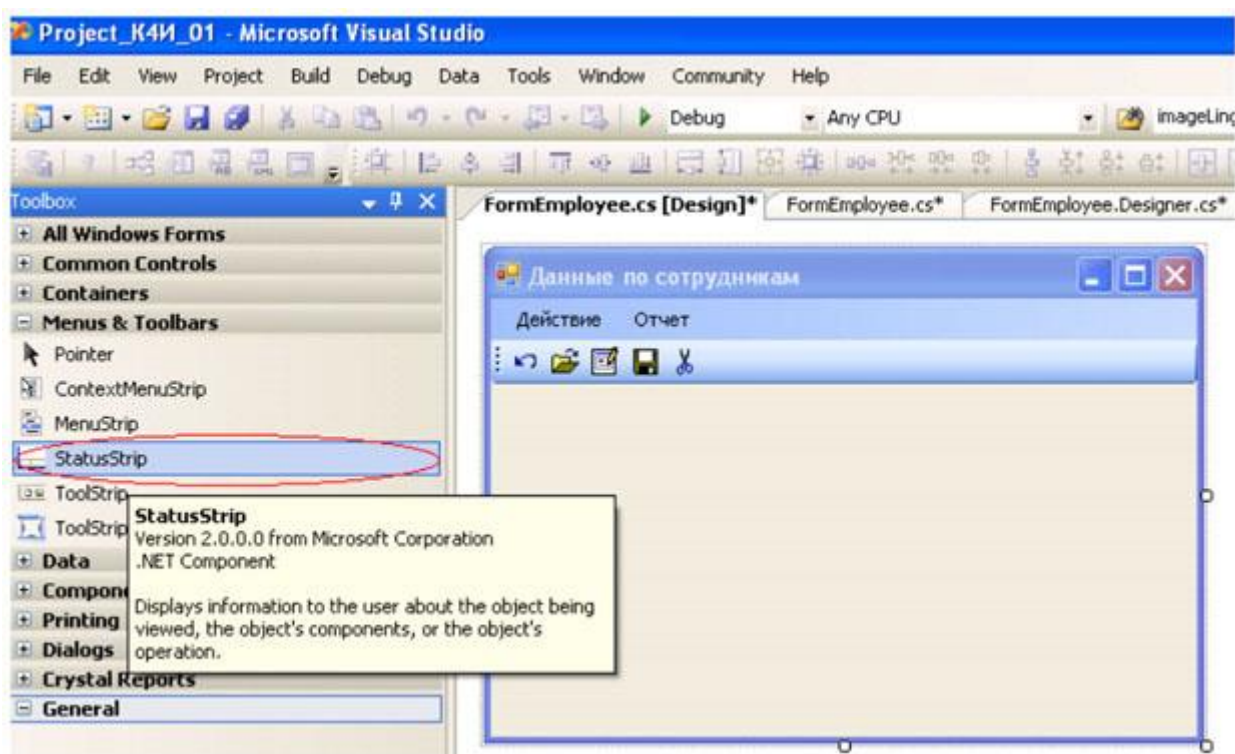
**Цель занятия:** Изучить основные способы построения строки состояния и получить практические навыки в разработке

### Создание строки состояния

На многих формах в реальных приложениях имеется элемент интерфейса, называемый строкой состояния (*StatusStrip*). Обычно в строке состояния выводится некоторая текстовая или графическая информация, относящаяся к работе приложения. Строка состояния может быть разделена на несколько "панелей" (*panel*) - отдельных частей окна. В каждой из этих панелей информация выводится отдельно.

Создадим строку состояния, в которой будут выводиться текстовые сообщения, относящиеся к пунктам *меню*.

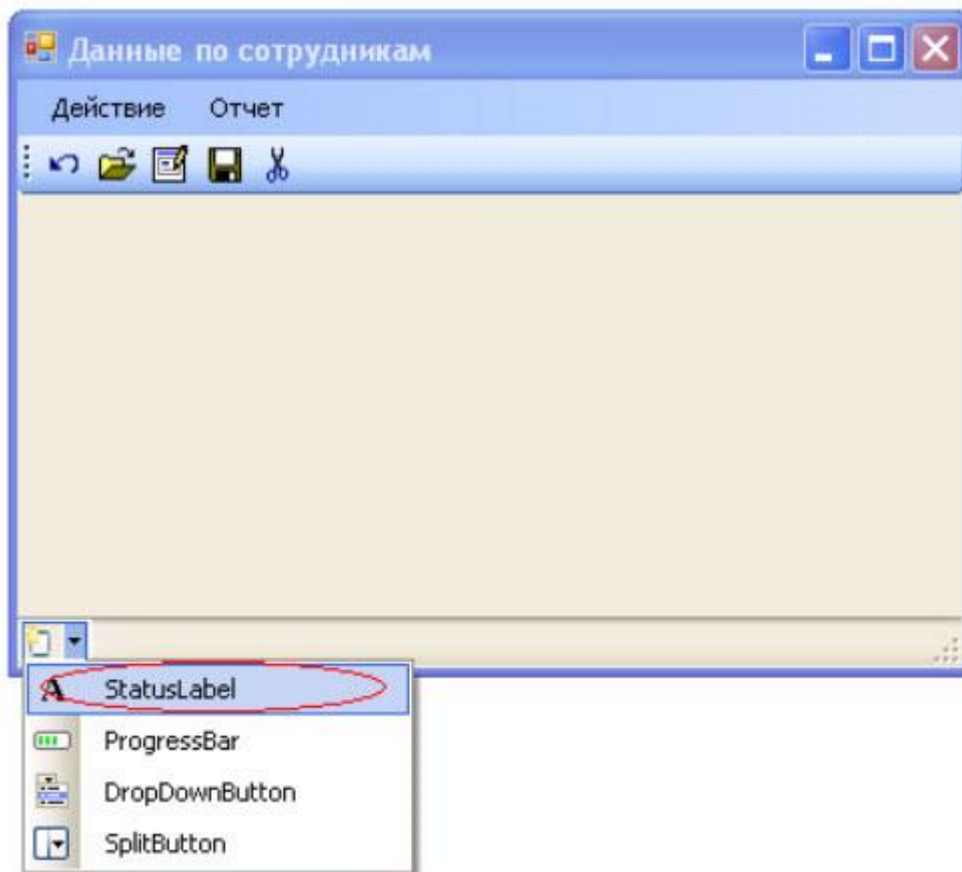
В окне *Toolbox* выделим пункт *StatusStrip* и перетащим его на форму ([рисунок 6.1](#)).



[увеличить изображение](#)

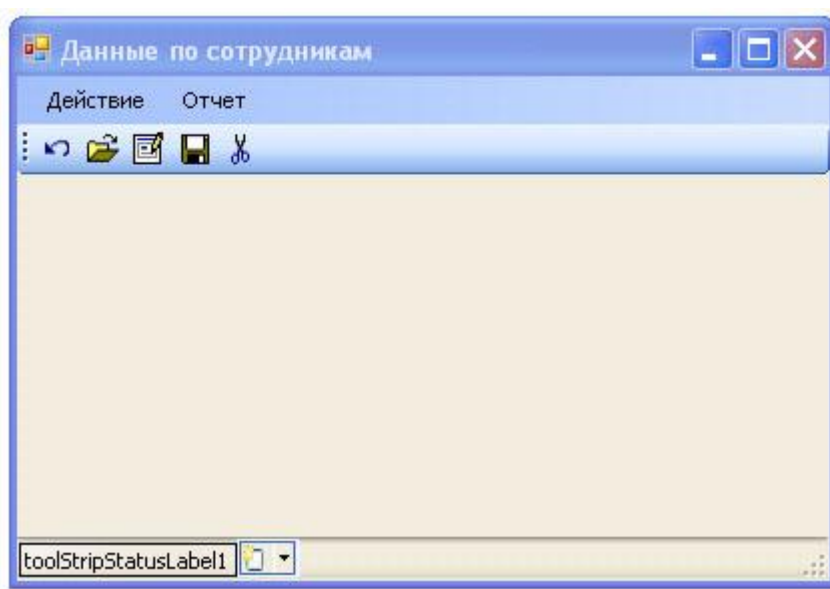
**Рис. 6.1.** Добавляем на форму строку состояния

Объекту класса *StatusStrip* присвоим имя *statusStripEmployee*. Откроем выпадающий список объекта класса *statusBarEmployee* и выберем объект *StatusLabel* ([рисунок 6.2](#)). Присвоим ему имя *toolStripStatusLabelEmployee*.



**Рис. 6.2.** Добавляем метку в строку состояния


При компиляции, запуске приложения и выборе пункта меню "Сотрудник" экранная форма будет иметь вид, представленный на [рисунке 6.3](#).

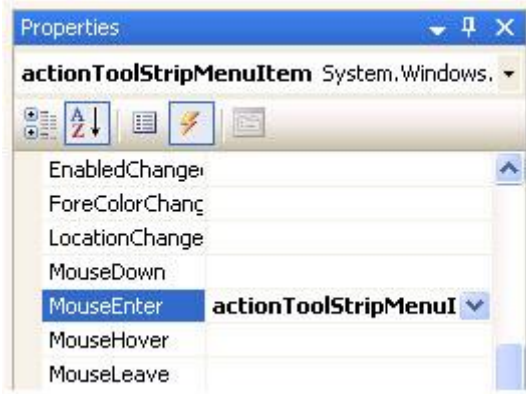


**Рис. 6.3.** Окно приложения со строкой состояния

Для управления текстом строки состояния необходимо разработать обработчик события для соответствующих объектов.

Для формы *FormEmployee* в строке состояний необходимо вывести информацию при наведении курсора мыши на пунктах меню "Действие". Первоначально в дизайнера формы необходимо

выделить пункт меню "Действие", перейти на вкладку *Properties* и открыть окно событий, нажав кнопку . На данной вкладке необходимо выделить событие `MouseEnter` и в поле ввода сделать двойной щелчок. (рисунок 6.4).



**Рис. 6.4.** Окно событий

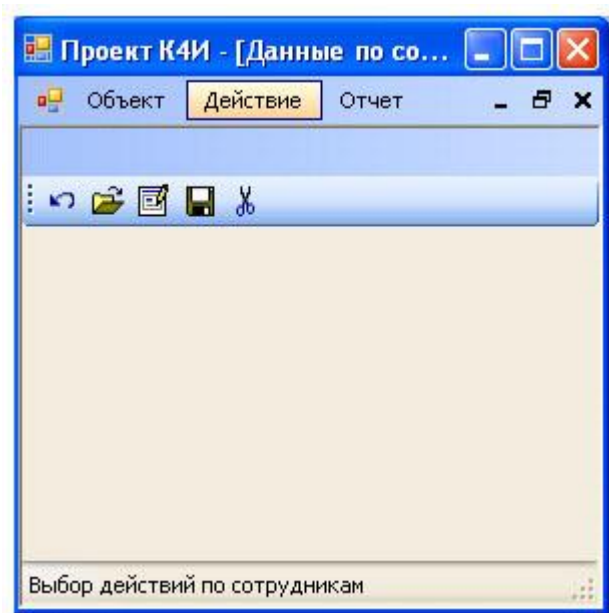
Система сгенерирует код обработчика, приведенного ниже.

```
private void actionToolStripMenuItem_MouseEnter(object sender, EventArgs e)
{ }
```

Добавим в обработчик следующий код:

```
private void actionToolStripMenuItem_MouseEnter(object sender, EventArgs e)
{
    toolStripStatusLabelEmployee.Text =
        "Выбор действий по сотрудникам";
}
```

Если откомпилировать программу, запустить её, выбрать пункт меню "Сотрудник" и навести указатель мыши на пункт "Действие", то сгенерируется событие "MouseEnter" и в строке состояния выведется текстовое сообщение "Выбор действий по сотрудникам" (рисунок 6.5).



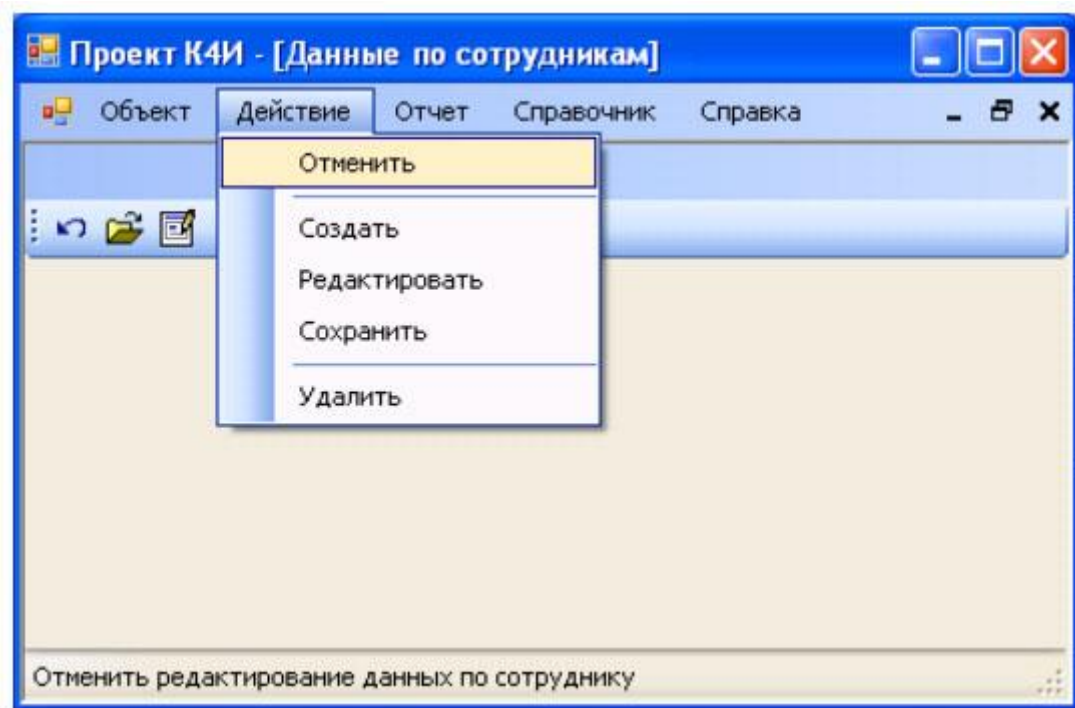
**Рис. 6.5.** Вывод сообщения в строке состояния

Если теперь переместить *указатель* мыши с пункта меню "*Действие*", то текст в строке состояния не изменится. Такой режим работы программы является неправильным, так как если *указатель* мыши перемещается с пункта меню "*Действие*", то строка состояния должна становиться пустой. Для обеспечения правильной работы программы воспользуемся ещё одним событием "*MouseLeave*", которое генерируется, когда *мышь* перемещается (покидает) с пункта меню "*Действие*". Обработчик данного события имеет следующий вид:

```
private void actionToolStripMenuItem_MouseLeave(object sender, EventArgs e)
{
    toolStripStatusLabelEmployee.Text = "";
}
```

Вышеприведенные обработчики будут вызываться только тогда, когда *пользователь* наведет *указатель* мыши на пункт меню "*Действие*". Для того чтобы обработчики реагировали на все строки пунктов главного меню "*Действие*" и "*Отчет*" формы *FormEmployee* необходимо сформировать соответствующие события *MouseEnter* и *MouseLeave* для всех подпунктов *меню* и создать для них обработчики.

Результаты компиляции и выполнения приложения приведены на [рисунке 6.6](#).



**Рис. 6.6.** Экранная форма со строкой состояния

### **Задание на практическое занятие**

1. Изучить теоретический материал.
2. Создать строку состояний для главной и дочерней форм.
3. Написать обработчики для формирования строки состояний, отображающих информацию о пунктах меню.
4. Протестировать работу приложения.

## Практическое занятие 7. Создание элементов управления

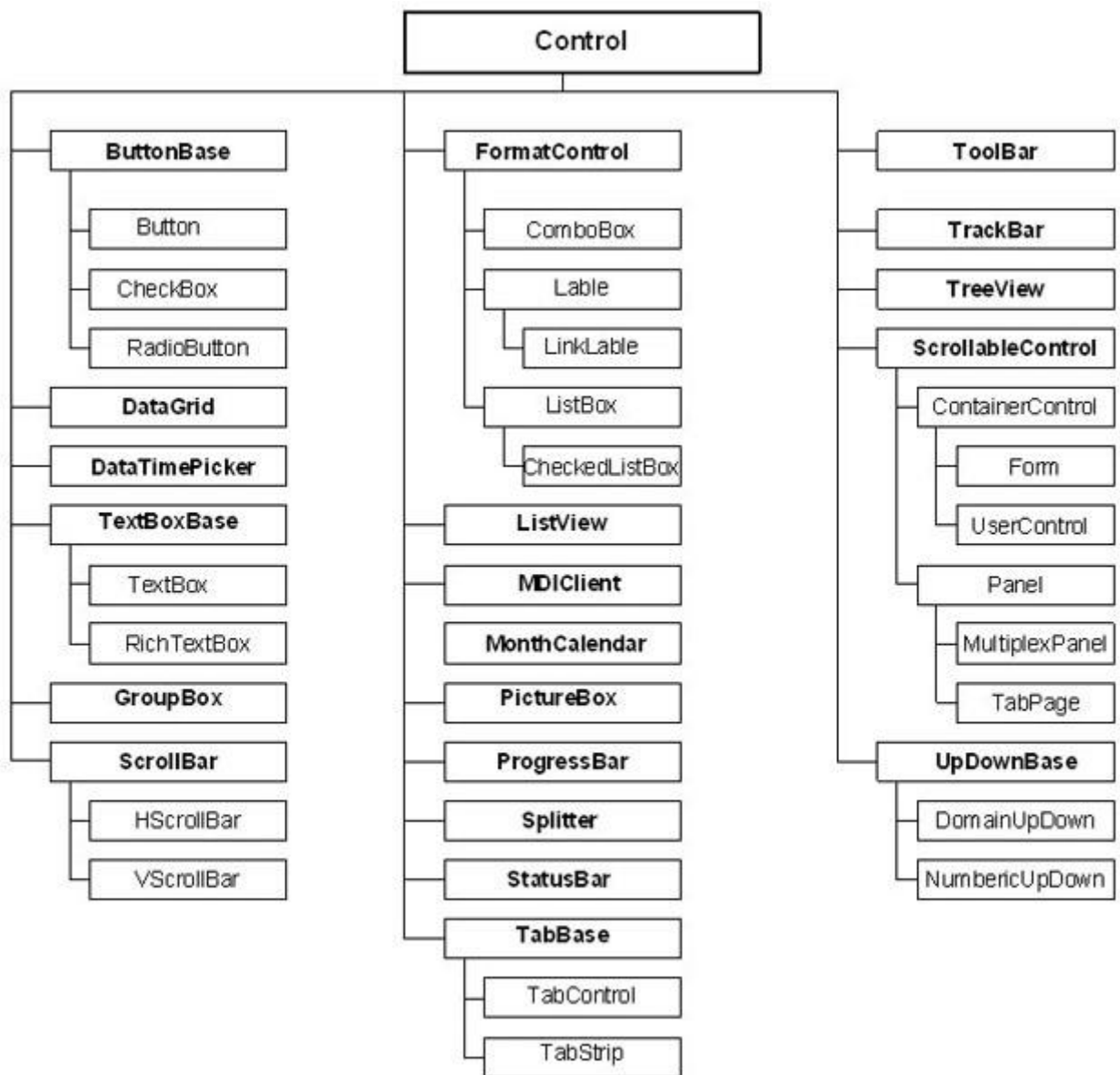
**Цель занятия:** Изучить основные *элементы управления Windows*-форм, их свойства и методы, а также получить практические навыки в разработке *Windows*-форм с элементами контроля

### 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 7.1](#).

*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 форму. *Visual Studio* автоматически сгенерирует нужный код для формы. После этого можно изменить название элемента управления на более содержательное (например, вместо *button1*, 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).



[увеличить изображение](#)


**Рис. 7.1.** Иерархия элементов управления Windows Forms

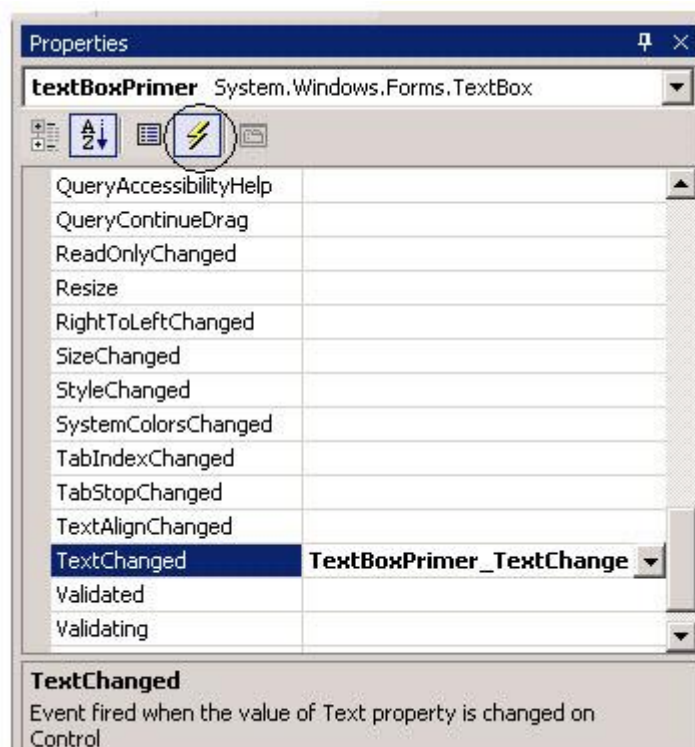
Все изменения, которые необходимо произвести в открывшемся окне ([рисунок 7.2](#)), будут добавлены в код метода `InitializeComponents()`.



**Рис. 7.2.** 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 7.3). Можно выбрать в списке нужное событие и рядом с ним сделать *двойной щелчок* или ввести имя метода или выбрать метод из списка.



**Рис. 7.3.** 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 и `RichTextBox`. Свойства, определенные в `TextBoxBase`, представлены в [таблице 7.1](#).

Свойство	Назначение
<code>AcceptsTab</code>	Определяет, что будет производиться при нажатии на клавишу <i>Tab</i> : вставка символа табуляции в само поле или переход к другому элементу управления
<code>AutoSize</code>	Определяет, будет ли элемент управления автоматически изменять размер при изменении шрифта на нем
<code>BackColor</code> , <code>ForeColor</code>	Позволяют получить или установить значение цвета фона и переднего плана
<code>HideSelection</code>	Позволяет получить или установить значение, определяющее, будет ли текст в <code>TextBox</code> оставаться выделенным после того, как этот элемент управления будет выведен из фокуса
<code>MaxLength</code>	Определяет максимальное количество символов, которое можно будет ввести в <code>TextBox</code>
<code>Modified</code>	Позволяет получить или установить значение, определяющее, был ли текст в <code>TextBox</code> изменен пользователем
<code>Multiline</code>	Указывает, может ли <code>TextBox</code> содержать несколько строк текста
<code>ReadOnly</code>	Помечает <code>TextBox</code> как "только для чтения"
<code>SelectedText</code> , <code>SelectionLength</code>	Содержат выделенный текст (или определенное количество символов) в <code>TextBox</code>

<code>SelectionStart</code>	Позволяет получить начало выделенного текста в <code>TextBox</code>
<code>Wordwrap</code>	Определяет, будет ли текст в <code>TextBox</code> 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 `TextChanged`. 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 7.2](#).

Таблица 7.2. Свойства, определенные в классе `TextBox`

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

Значения перечисления `HorizontalAlignment` представлены в [таблице 7.3](#).

Таблица 7.3. Значения перечисления HorizontalAlignment

Значение	Описание
<code>Center</code>	Выравнивание по центру
<code>Left</code>	Выравнивание по левому краю
<code>Right</code>	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 7.4](#).

Таблица 7.4. Свойства ButtonBase

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

<code>TextAlign</code>	Позволяет получить или установить выравнивание текста на кнопке. Также используются значения из перечисления <code>ContentAlignment</code>
------------------------	--

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 7.5](#)). Значения из того же перечисления используются и для определения положения изображения на кнопке.

Таблица 7.5. Значения перечисления <code>ContentAlignment</code>	
Значение	Описание (выравнивание)
<code>BottomCenter</code>	По нижнему краю кнопки, относительно боковых краев - посередине
<code>BottomLeft</code>	По нижнему краю кнопки, слева
<code>BottomRight</code>	По нижнему краю кнопки, справа
<code>MiddleCenter</code>	По центру кнопки
<code>MiddleLeft</code>	Относительно верхнего и нижнего краев - по центру, относительно боковых краев - слева
<code>MiddleRight</code>	Относительно верхнего и нижнего краев - по центру, относительно боковых краев - справа
<code>TopCenter</code>	По верхнему краю кнопки, относительно боковых краев - посередине
<code>TopLeft</code>	По верхнему краю кнопки, слева
<code>TopRight</code>	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 7.6](#).

Таблица 7.6. Свойства класса `CheckBox`

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

Возможные состояния флажка (`Indeterminate` можно использовать только тогда, когда для свойства `ThreeState` установлено значение `true`) представлены в [таблице 7.7](#).

Таблица 7.7. Значения перечисления `CheckState`

Значение	Описание
<code>Checked</code>	Флажок установлен
<code>Indeterminate</code>	Значение не определено (обычно флажок выглядит как "серый", затененный)
<code>Unchecked</code>	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 тип `GroupBox`.

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 `Checked` использовать 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) и `ComboBox` (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 класса `ComboBox`. Наиболее важные свойства `System.Windows.Forms.ListBox` представлены в [таблице 7.8](#).

Таблица 7.8. Свойства класса `ListBox`

Свойство	Назначение
<code>ScrollAlwaysVisible</code>	Определяет, будет ли полоса прокрутки выводиться всегда
<code>SelectedIndex</code>	Индекс выделенного в настоящий момент элемента в списке (если такой имеется). Если ни один элемент не выделен, то возвращается значение <code>-1</code>
<code>SelectedIndices</code>	Набор индексов выделенных в настоящий момент элементов в списке. Если не выделен ни один элемент, то возвращается пустой набор
<code>SelectedItem</code>	Значение выделенного в настоящий момент элемента. Если ни один из элементов не выделен, то возвращается <code>null</code>
<code>SelectedItems</code>	Возвращает коллекцию значений выделенных элементов (для списков, в которых допускается выбор нескольких значений)
<code>SelectionMode</code>	Определяет число элементов, которые возможно выбрать в списке одновременно. Для этого свойства используются значения из перечисления <code>SelectionMode</code>
<code>Sorted</code>	Определяет, будут ли элементы в списке упорядочены (по алфавиту) или нет
<code>TopIndex</code>	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 7.9](#).

Таблица 7.9. Свойства `ComboBox`

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

Стиль для `ComboBox` можно настроить при помощи свойства `Style`, для которого используются значения из перечисления `ComboBoxStyle` ([таблица 7.10](#)).

Таблица 7.10. Значения перечисления `ComboBoxStyle`

Значение	Описание
----------	----------

<code>DropDown</code>	Пользователь может вводить значения в поле редактирования. Для отображения списка пользователь должен нажать на кнопку со стрелкой, направленной вниз ( <code>Arrow Button</code> )
<code>DropDownList</code>	Пользователь не может вводить значения в поле редактирования. Для отображения списка пользователь должен нажать на кнопку со стрелкой, направленной вниз ( <code>Arrow Button</code> )
<code>Simple</code>	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 `TabIndex`. 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 `TabIndex`:

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 дружелюбный и удобный интерфейс. Это элемент управления `MonthCalendar`.

Наиболее важные свойства `MonthCalendar` представлены в [табл. 7.11](#).

Таблица 7.11. Свойства MonthCalendar	
Свойство	Назначение
<code>BoldedDates</code>	Массив объектов <code>DateTime</code> , выделенных подсветкой
<code>CalendarDimensions</code>	Определяет количество выводимых строк и столбцов

<code>FirstDayOfWeek</code>	Определяет, с какого дня будет начинаться неделя в <code>MonthCalendar</code>
<code>MaxDate</code>	Самая поздняя дата, которую разрешается выбрать пользователю (по умолчанию ограничений нет)
<code>MaxSelectionCount</code>	Максимальное количество дат, которое одновременно может выбрать пользователь
<code>MinDate</code>	Самая ранняя дата, которую разрешается выбрать пользователю (по умолчанию ограничений нет)
<code>MonthlyBoldedDates</code>	Массив выделенных подсветкой объектов <code>DateTime</code> для месяца
<code>SelectionRange</code>	Диапазон выделенных объектов
<code>SelectionEnd</code>	Самая поздняя дата в диапазоне выделенных объектов
<code>SelectionStart</code>	Самая ранняя дата в диапазоне выделенных объектов
<code>ShowToday</code>	Определяет, будет ли <code>MonthCalendar</code> выводить информацию о текущей дате
<code>ShowTodayCircle</code>	Определяет, будет ли <code>MonthCalendar</code> выводить информацию о текущей дате в нижней части и выделять ее в календаре обводкой
<code>ShowWeekNumbers</code>	Определяет, будет ли <code>MonthCalendar</code> отображать номера недель справа от каждой строки
<code>TodayDate</code>	Дата, которая будет считаться <code>MonthCalendar</code> сегодняшней. По умолчанию <code>TodayDate</code> - это системная дата на момент создания объекта <code>MonthCalendar</code>
<code>TodayDateSet</code>	Определяет, можно ли пользователю по своему усмотрению выбирать сегодняшнюю дату. Если для этого свойства установлено значение <code>true</code> , 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 типа `DateTime` 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 из `DateTime` при помощи специальных "форматирующих" свойств самих объектов `DateTime`. Набор таких свойств (и некоторые методы) представлен в [таблице 7.12](#).

Таблица 7.12. Члены класса `DateTime`

Член	Назначение
<code>Date</code>	Позволяет получить информацию о дате (дата всегда отсчитывается от полуночи)
<code>Day</code> , <code>Month</code> , <code>Year</code>	Позволяют получить соответственно день, месяц и число из текущего объекта <code>DateTime</code>
<code>DayOfWeek</code>	Возвращает день недели для объекта <code>DateTime</code>
<code>DayOfYear</code>	Возвращает номер дня в году для объекта <code>DateTime</code>
<code>Hour</code> , <code>Minute</code> , <code>Second</code> , <code>Millisecond</code>	Возвращают информацию о часе, минутах, секундах и миллисекундах для объекта <code>DateTime</code>
<code>MaxValue</code> , <code>MinValue</code>	Возвращают минимальное и максимальное значения для <code>DateTime</code>
<code>Now</code> , <code>Today</code>	Эти два статических свойства типа <code>DateTime</code> позволяют получить информацию о текущей дате и времени ( <code>Now</code> ) или только о текущей дате ( <code>Today</code> )

<code>Ticks</code>	Позволяет получить счетчик "тиков" (с интервалом в 100 наносекунд) для объекта <code>DateTime</code>
<code>ToLongDateString( )</code> , <code>ToLongTimeString( )</code> , <code>ToShortDateString( )</code> , <code>ToShortTimeString( )</code>	Преобразуют текущее значение объекта <code>DateTime</code> 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 свойства `AutoScroll` объекта `Panel` значение `true`. В результате пользователь получит возможность доступа к "не вмещающимся" элементам управления с помощью полос прокрутки.

## Всплывающие подсказки (ToolTip)

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 7.13](#).

Таблица 7.13. Члены класса `ToolTip`

Член	Назначение
<code>Active</code>	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
<code>AutomaticDelay</code>	Позволяет получить или установить время задержки (в миллисекундах) при появлении подсказки
<code>AutoPopDelay</code>	Время (в миллисекундах), в течение которого подсказка остается видимой, если указатель мыши неподвижен и находится в области, занимаемой соответствующим элементом управления. По умолчанию

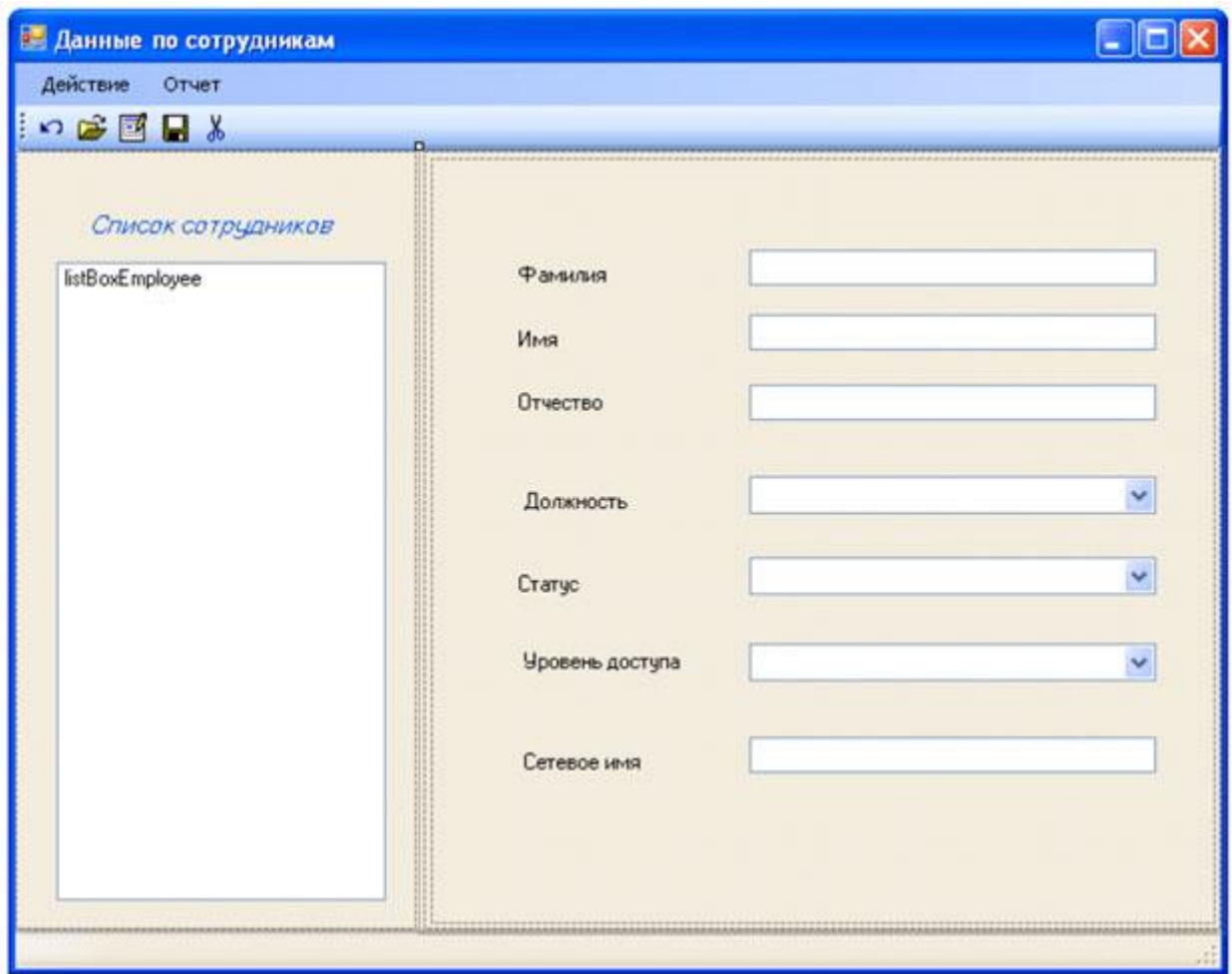
	это значение равно 10 значениям <code>AutomaticDelay</code>
<code>GetTooltip( )</code>	Возвращает текст подсказки
<code>InitialDelay</code>	Время (в миллисекундах), в течение которого указатель должен оставаться неподвижным в соответствующей области для появления подсказки. Значение по умолчанию равно значению <code>AutomaticDelay</code>
<code>ReshowDelay</code>	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 <code>AutomaticDelay</code>
<code>SetToolTip( )</code>	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 7.4](#).



[увеличить изображение](#)

**Рис. 7.4.** Вид экранной формы FormEmployee

На данной форме необходимо сформировать *элементы управления*, приведенные в [таблице 7.14](#).

Таблица 7.14. Элементы управления формы FormEmployee

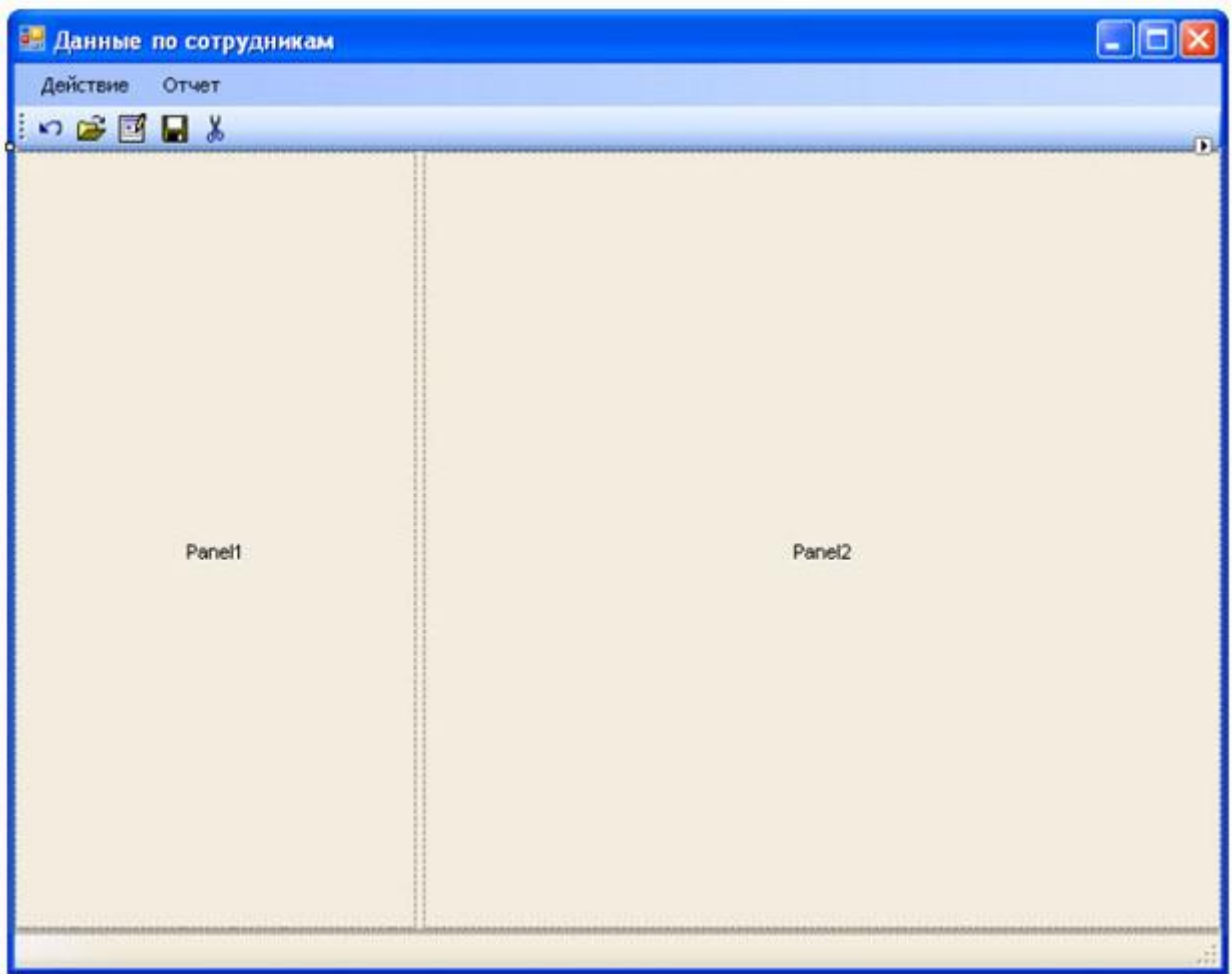
Элемент контроля	Имя	Свойство Text/Items	Назначение
SplitContainer	splitContainerEmployee		Две панели с разделителем
Label	labelListEmployee	Список сотрудников	Надпись
listBox	listBoxEmployee		Список сотрудников

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"
<code>menuItem</code>	<code>menuItemUndo</code>	Отменить	Подпункт меню "Отменить"
<code>menuItem</code>	<code>menuItemNew</code>	Создать	Подпункт меню "Новый"
<code>menuItem</code>	<code>menuItemEdit</code>	Изменить	Подпункт меню "Изменить"
<code>menuItem</code>	<code>menuItemSave</code>	Сохранить	Подпункт меню "Сохранить"
<code>menuItem</code>	<code>menuItem</code>	Удалить	Подпункт меню "Удалить"
<code>menuItem</code>	<code>menuItemReport</code>	Отчет	Пункт меню "Отчет"
<code>menuItem</code>	<code>menuItemReport1</code>	По сотруднику	Подпункт меню "Отчет по сотруднику"
<code>menuItem</code>	<code>menuItemReport2</code>	По всем сотрудникам	Подпункт меню "Отчет по всем сотрудникам"

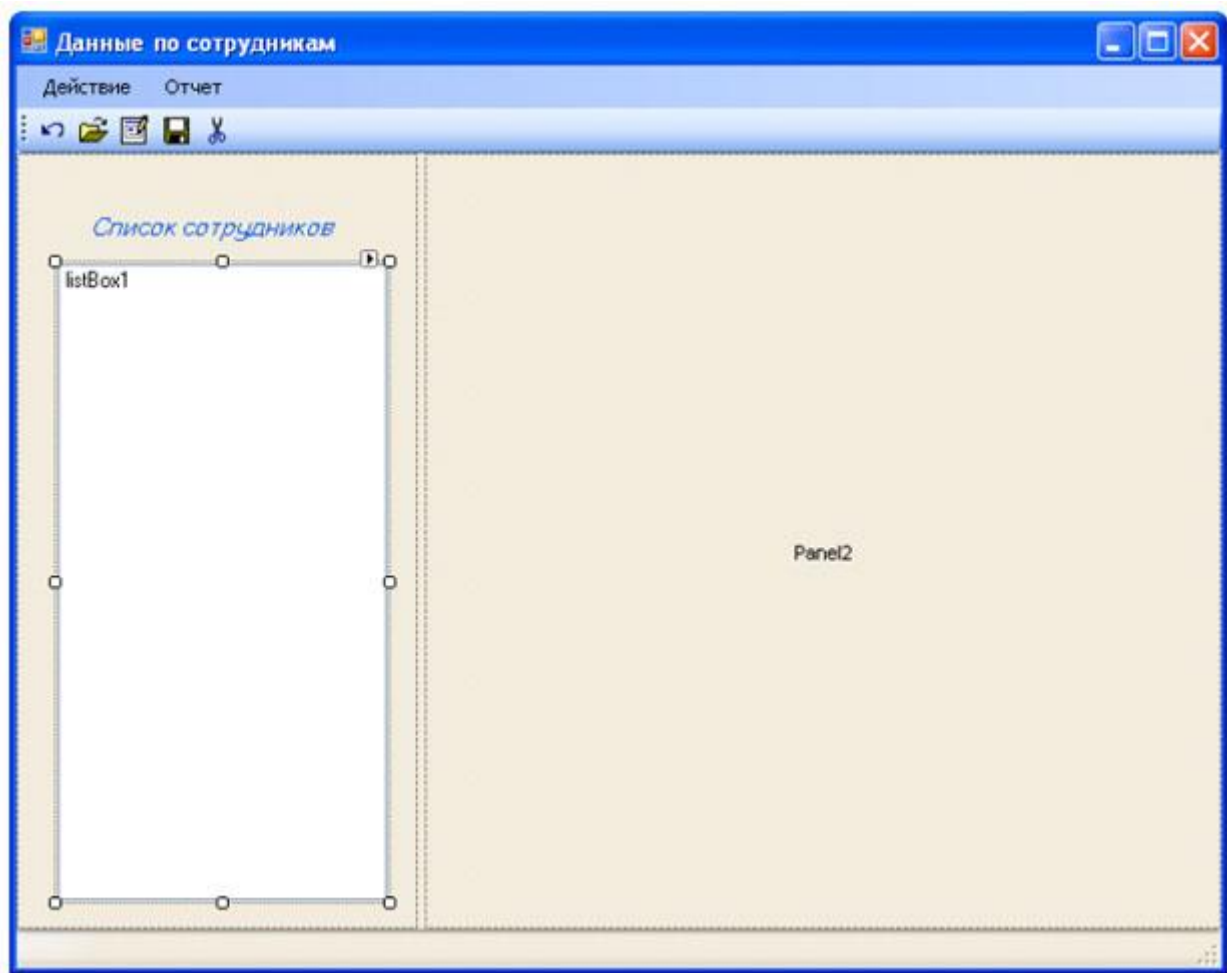
Вначале создайте на форме элемент *SplitContainer* ([рисунок 7.5](#)). На панели 1 создайте элементы управления *labelListEmployee* и *listBoxEmployee* ([рисунок 7.6](#)), а остальные элементы управления, приведенные в [таблице 7.14](#), - на панели 2 ([рисунок 7.4](#)).

После создания на форме *FormEmployee* элементов управления в соответствии с [таблицей 7.14](#) необходимо настроить порядок перехода между ними при нажатии клавиши *Tab*.



[увеличить изображение](#)

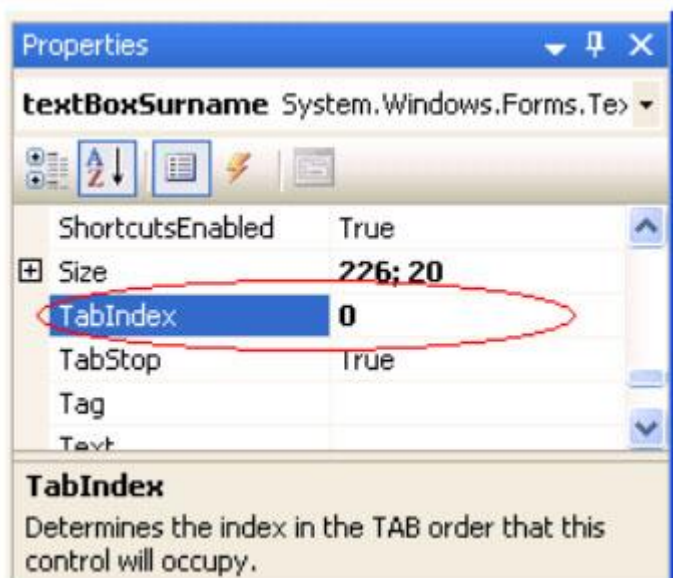
**Рис. 7.5.** Создание панелей на форме FormEmployee



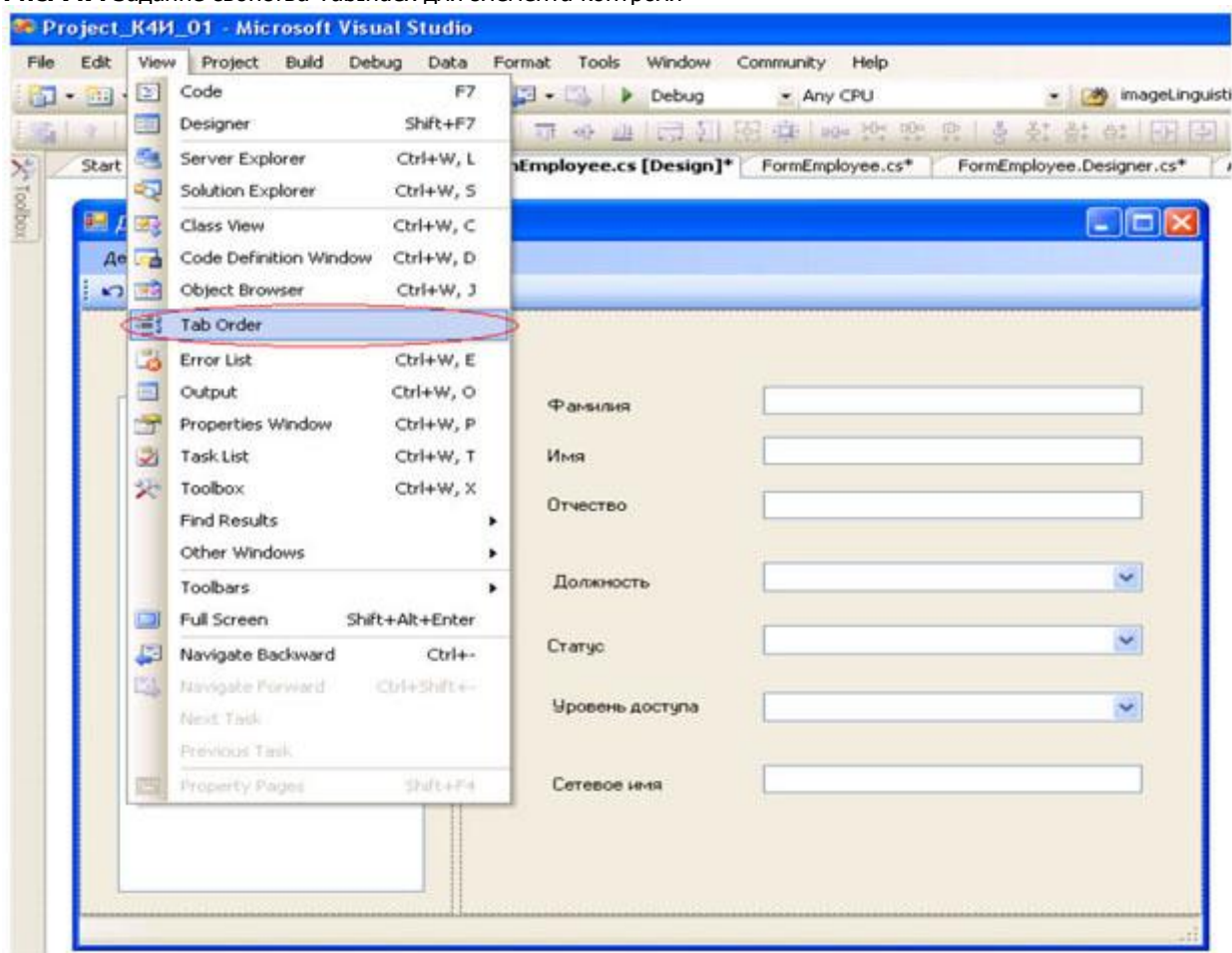
[увеличить изображение](#)

**Рис. 7.6.** 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 7.7](#)) или вызвать мастер `Tab Order Wizard` из меню *View/Tab Order* ([рисунок 7.8](#)). Задание последовательности значений свойству `TabIndex` производится щелчком мыши на элементах управления в заданной последовательности.



**Рис. 7.7.** Задание свойства TabIndex для элемента контроля

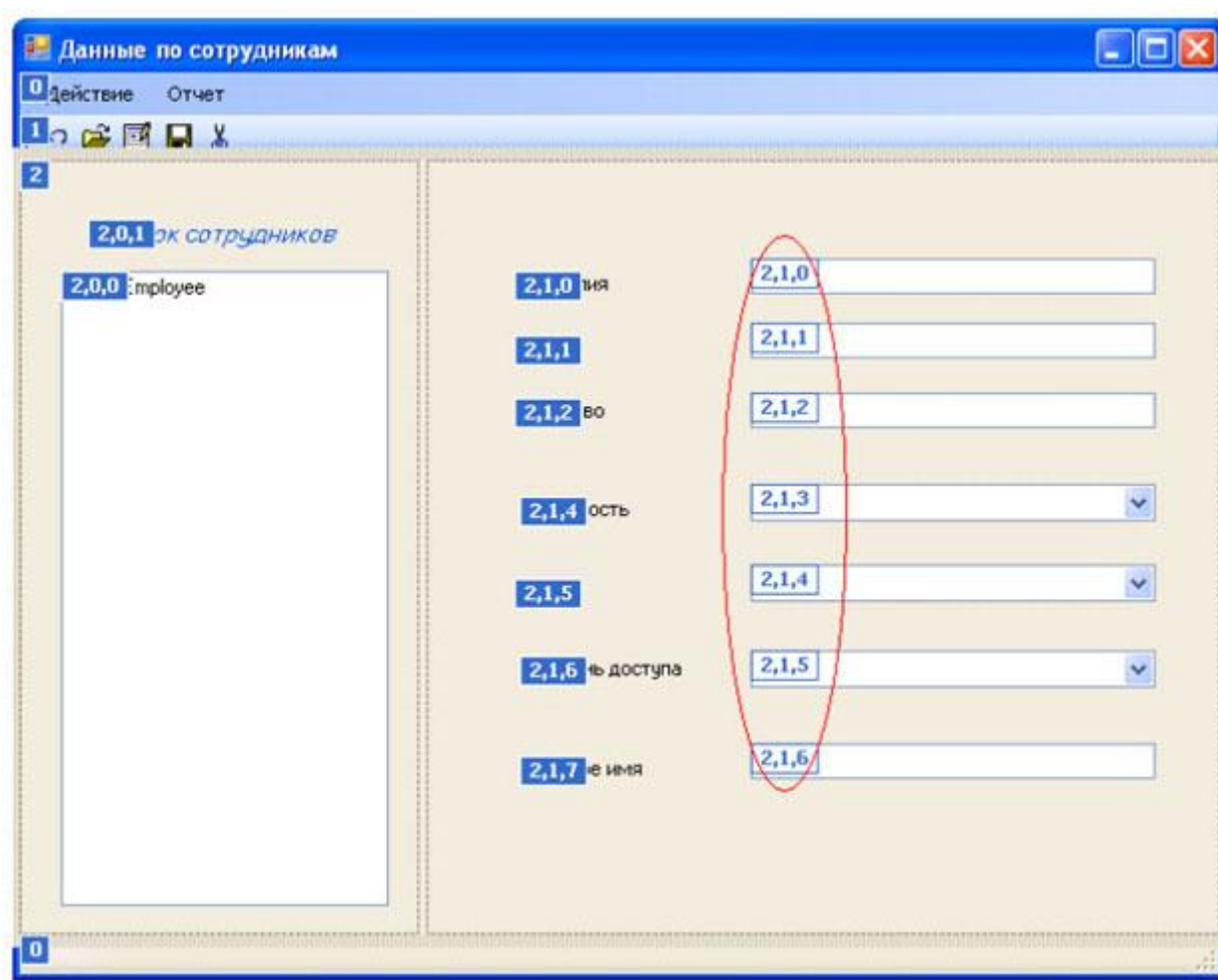


[увеличить изображение](#)

**Рис. 7.8.** Настройка перехода по элементам управления

Результат настройки порядка перехода между элементами управления при нажатии клавиши *Tab* приведен на [рисунок 7.9](#).

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



[увеличить изображение](#)

**Рис. 7.9.** 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 `TextBox` необходимо 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:

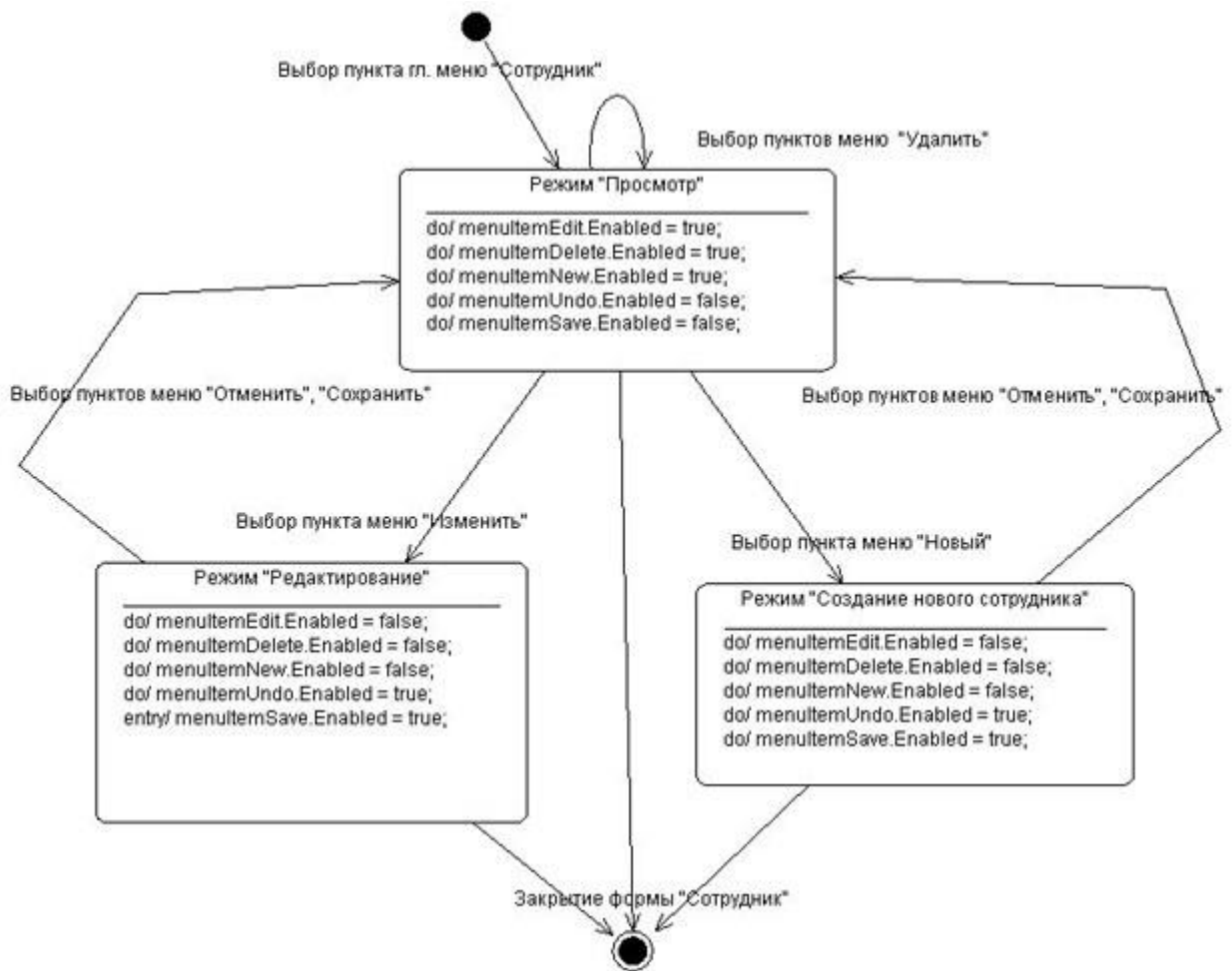
```
/// Задание режима редактирования
public void DisplayEdit()
{
    this.textBoxSurname.ReadOnly = false;
    this.textBoxName.ReadOnly = false;
    this.textBoxPatronymic.ReadOnly = false;
    this.textBoxNetName.ReadOnly = false;
    this.comboBoxJobRole.Enabled = true;
    this.comboBoxStatus.Enabled = true;
    this.comboBoxAccess.Enabled = true;
}
```

Для управления режимом доступности (только для чтения/редактирование) формы *FormEmployee* необходимо метод `DisplayReadOnly` 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 параметр `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 7.10](#).



[увеличить изображение](#)

**Рис. 7.10.** 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 7.10 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` класса `ToolStripMenuItem`, а кнопка 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 практическое занятие

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.

## Практическое занятие 8. Подготовка ADO.NET к работе в приложении

**Цель занятия:** 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.

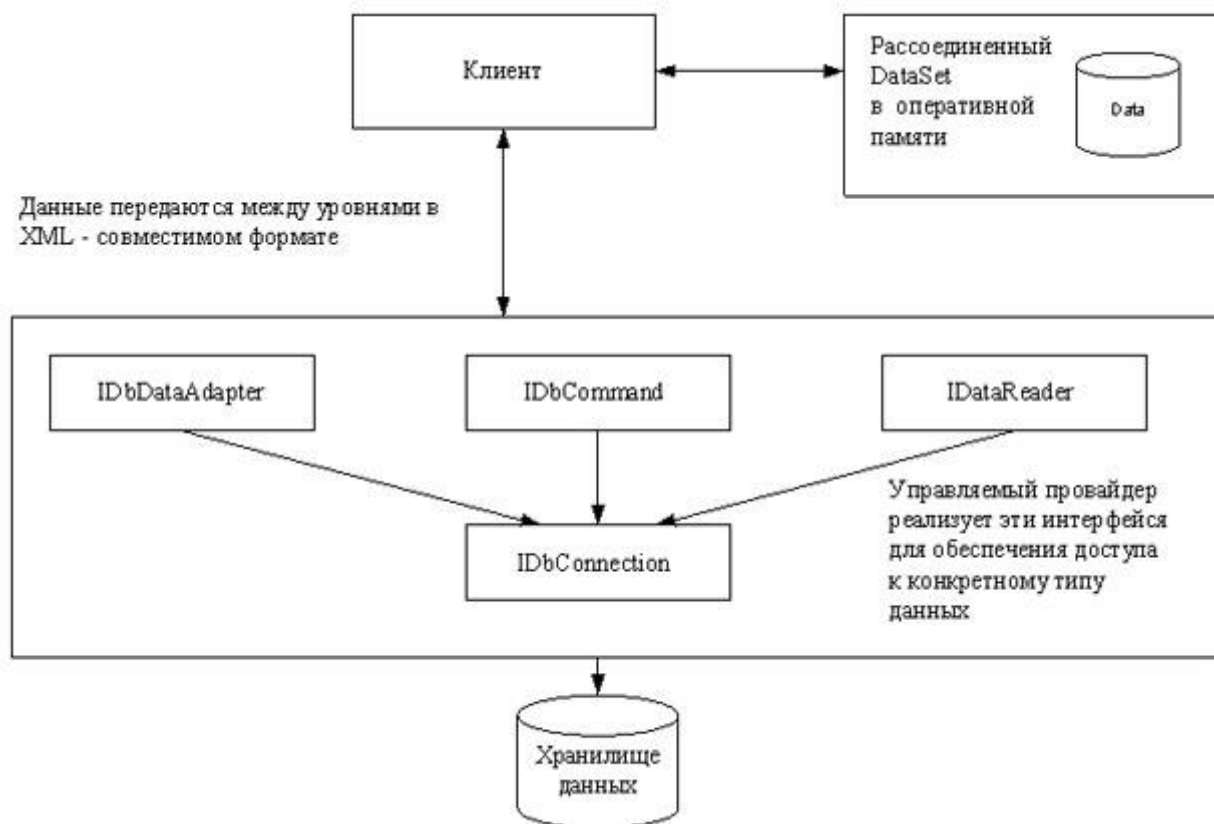
*DataSet* - набор связанных таблиц. На практике можно создать на клиенте объект *DataSet*, который будет представлять полную копию удаленной базы данных.

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* (рисунки 8.1).



[увеличить изображение](#)

**Рис. 8.1.** 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 8.1](#).

Таблица 8.1. 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.SqlClient	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 8.2](#).

Таблица 8.2. Типы пространства имен System.Data

Тип	Назначение
<i>DataColumnCollection</i> , <i>DataColumn</i>	<i>DataColumn</i> представляет один столбец в объекте <i>DataTable</i> , <i>DataColumnCollection</i> - все столбцы
<i>ConstraintCollection</i> , <i>Constraint</i>	<i>Constraint</i> - объектно-ориентированная оболочка вокруг ограничения (например, внешнего ключа или уникальности), наложенного на один или несколько <i>DataColumn</i> , <i>ConstraintCollection</i> - все ограничения в объекте <i>DataTable</i>

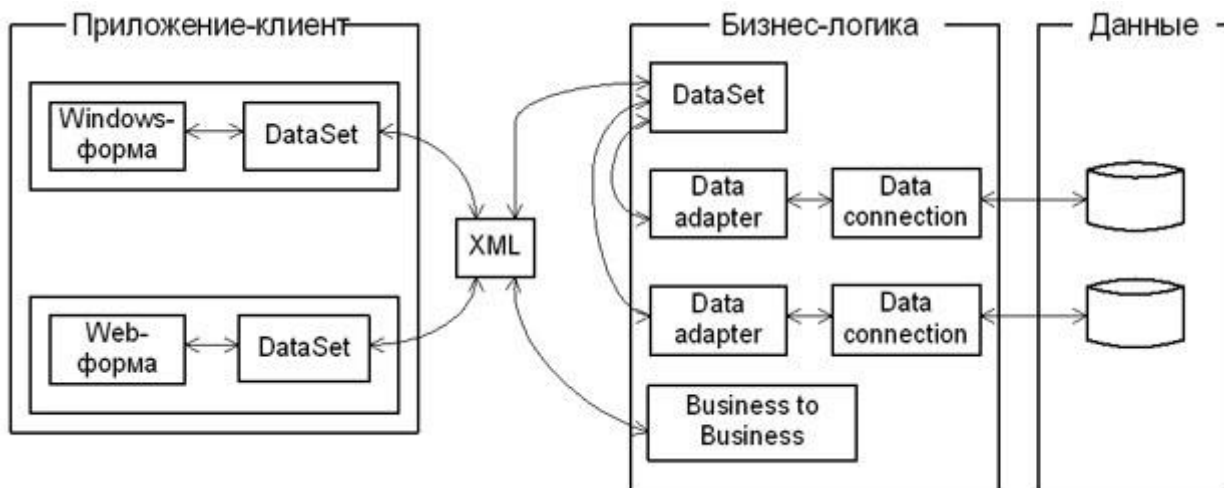
<code>DataRowCollection</code> , <code>Data Row</code>	<code>DataRow</code> представляет единственную строку в <code>DataTable</code> , <code>DataRowCollection</code> - все строки в <code>DataTable</code>
<code>DataRowView</code> , <code>Data View</code>	<code>DataRowView</code> позволяет создавать настроенное представление единственной строки, <code>DataRowView</code> - созданное программным образом представление объекта <code>DataRow</code> , которое может быть использовано для сортировки, фильтрации, поиска, редактирования и перемещения
<code>DataSet</code>	Объект, создаваемый в оперативной памяти на клиентском компьютере. <code>DataSet</code> состоит из множества объектов <code>DataTable</code> и информации об отношениях между ними
<code>ForeignKeyConstraint</code> , <code>UniqueConstraint</code>	<code>ForeignKeyConstraint</code> представляет ограничение, налагаемое на набор столбцов в таблицах, связанных отношениями первичный - внешний ключ. <code>UniqueConstraint</code> - ограничение, при помощи которого гарантируется, что в столбце не будет повторяющихся записей
<code>DataRelationCollection</code> , <code>DataRelation</code> , <code>DataTableCollection</code> , <code>DataTable</code>	Тип <code>DataRelationCollection</code> представляет набор всех отношений (то есть объектов <code>DataRelation</code> ) между таблицами в <code>DataSet</code> . Тип <code>DataTableCollection</code> представляет набор всех таблиц (объектов <code>DataTable</code> ) в <code>DataSet</code>

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 освободить занимаемые серверные ресурсы, что обеспечит обслуживание большего количества пользователей. Модели доступа к данным представлена на [рисунке 8.2](#).



**Рис. 8.2.** 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 *SELECT* или 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 содержится *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.

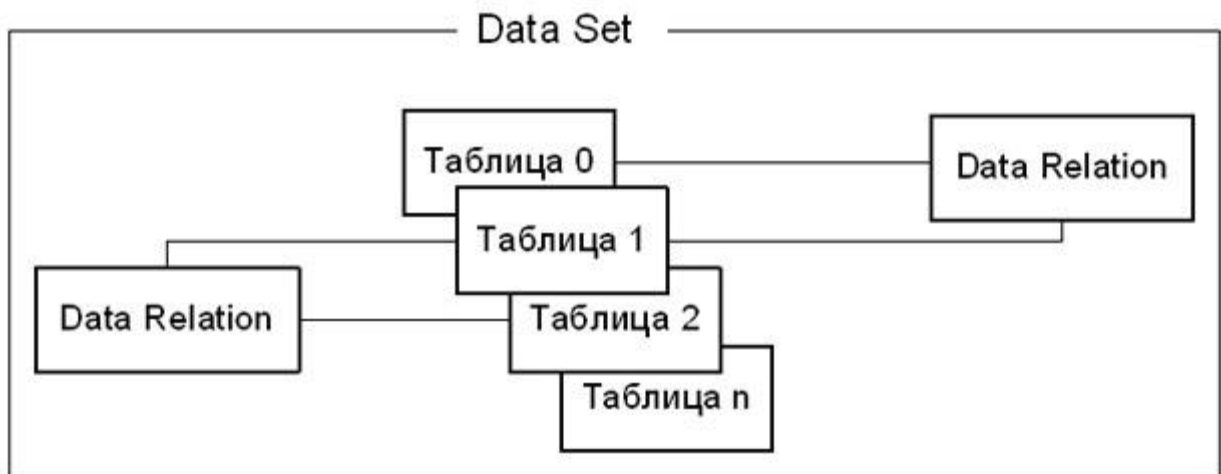
Команды работы с данными могут содержать параметры, т. е. могут использоваться параметризованные запросы, как, например, следующий *запрос*:

```
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 8.3](#).



**Рис. 8.3.** 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.

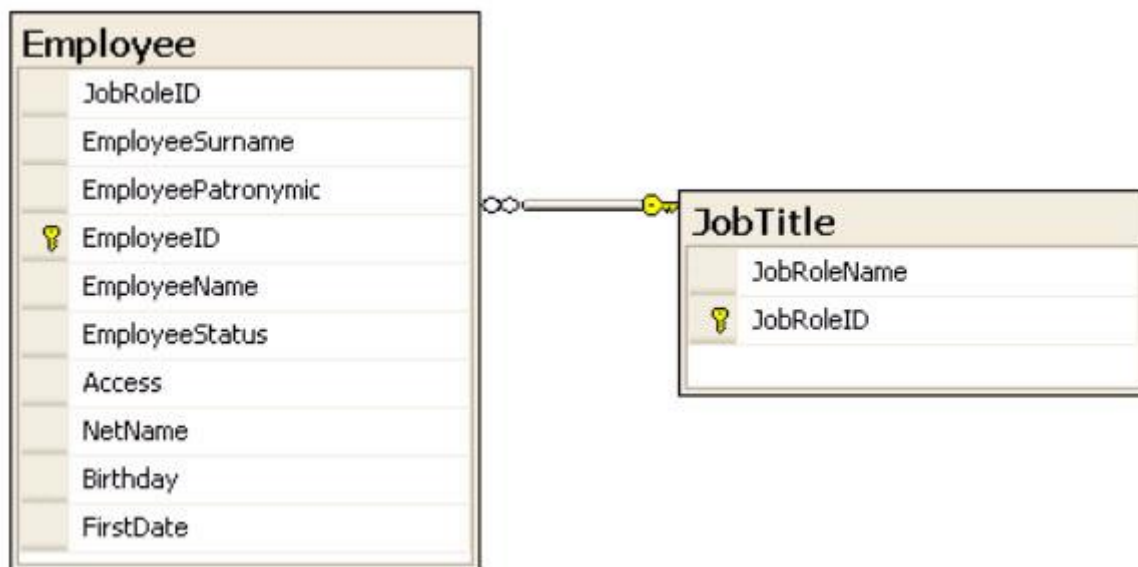
Например, метод *Fill* объекта *DataAdapter*, заполняющего данными *контейнер DataSet*, может использовать в элементе *SelectCommand* следующий *запрос*:

```
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 8.4](#) представлена структура *базы данных*.



**Рис. 8.4.** 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 8.4](#)

Таблица 8.4. 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

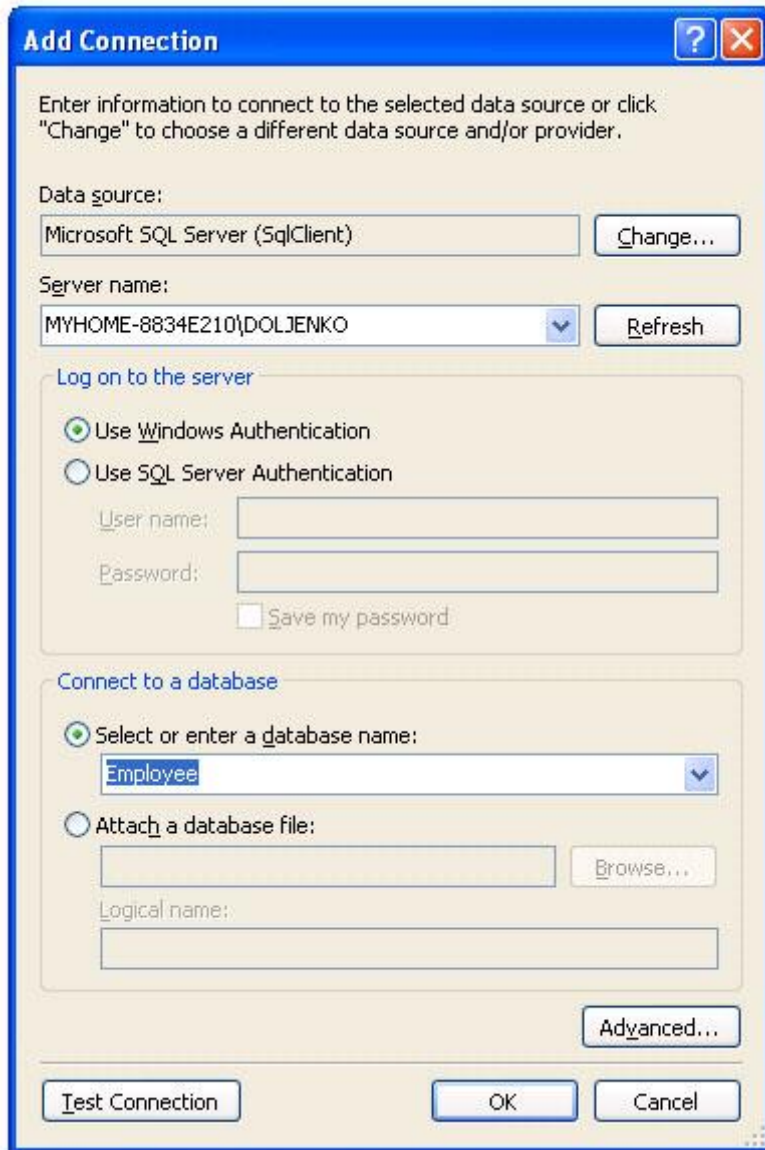
Access	Уровень доступа	varchar (20)
NetName	Сетевое имя	varchar (20)
Birthday	Дата рождения	Smalldatetime
FirstDate	Дата приема на работу	smalldatetime

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 8.3](#).

Таблица 8.3. Атрибуты таблицы JobTitle		
Имя атрибута	Назначение	Тип
JobRoleID	Суррогатный ключ	smallint
JobRoleName	Наименование должности	varchar (50)

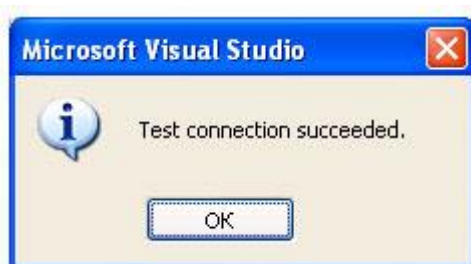
В рассматриваемом приложении в качестве *СУБД* используется *MS SQL Server 2005*. Создаем соединение проекта с базой данных. Для этого выбираем пункт меню *Tools/Connect to Database*. Появляется окно *AddConnection* ([рисунок 8.5](#))



**Рис. 8.5.** Окно AddConnection

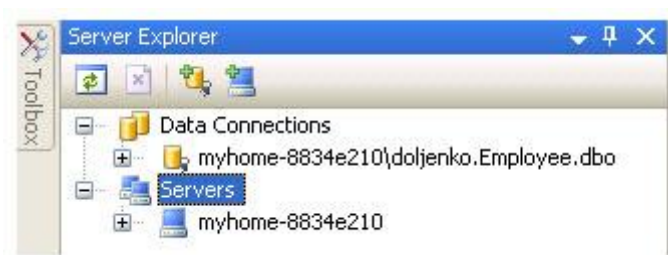
В пункте *"Server name"* задаем имя сервера, которое необходимо узнать у преподавателя (на [рисунке 8.5](#) `MYHOME-8834E210\DOLJENKO`). В пункте *Select or enter database name* - имя *базы данных*, которое определит преподаватель (на [рисунке 8.5](#) - `Employee`).

Для проверки правильности подключения к базе данных нажимаем клавишу *"Test Connection"*. При правильном подключении появляется следующее сообщение ([рисунк 8.6](#)).



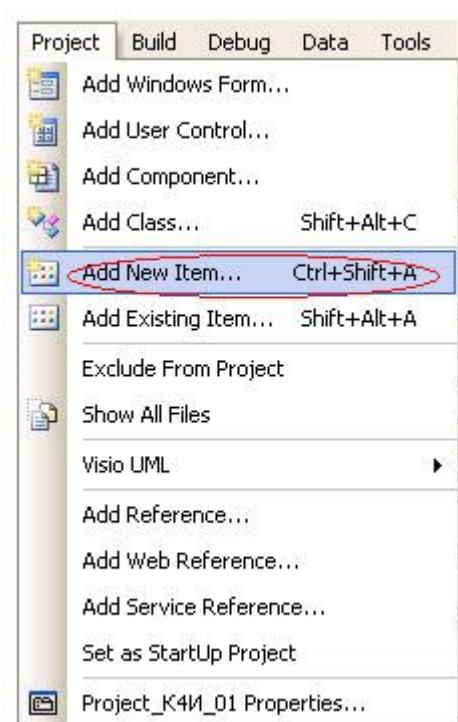
**Рис. 8.6.** Окно Microsoft Data Link

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



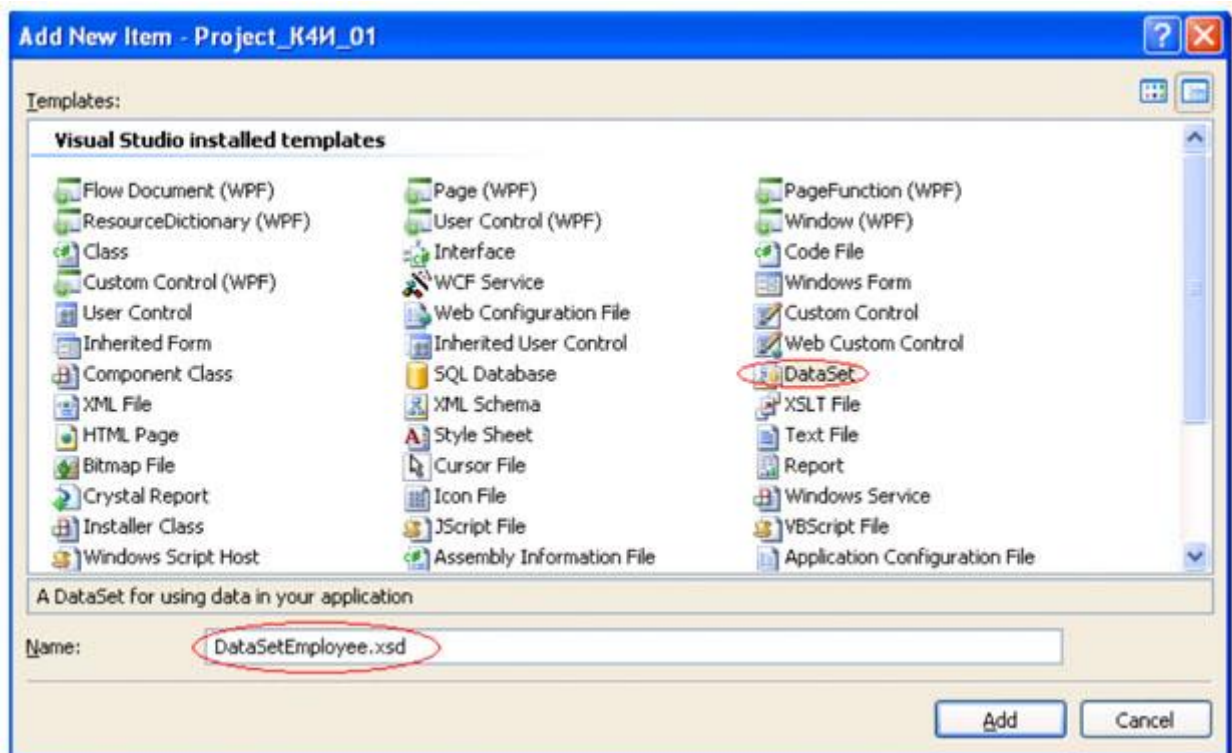
**Рис. 8.7.** Окно навигатора Server Explorer

Добавим в проект *объект* класса *DataSet*. Для этого выберем *пункт меню Project/Add New Item. . .* ([рисунк 8.8](#)).



**Рис. 8.8.** Добавление в проект нового компонента

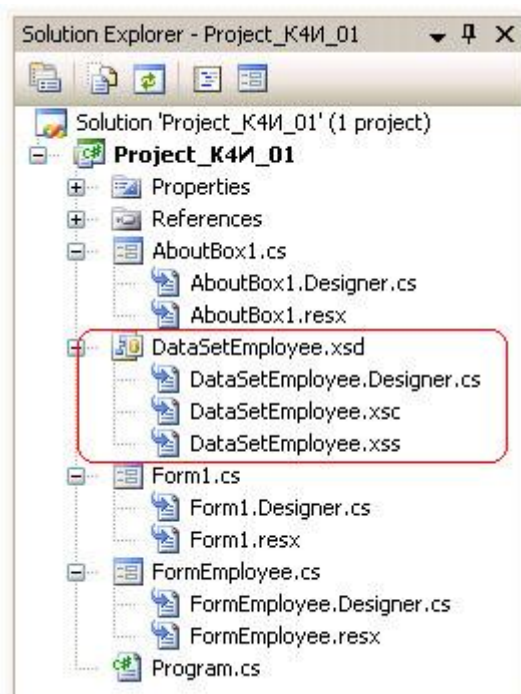
В окне *Add New Item* ([рисунк 8.9](#)) выберем *шаблон DataSet* и присвоим ему имя *DataSetEmployee*.



[увеличить изображение](#)

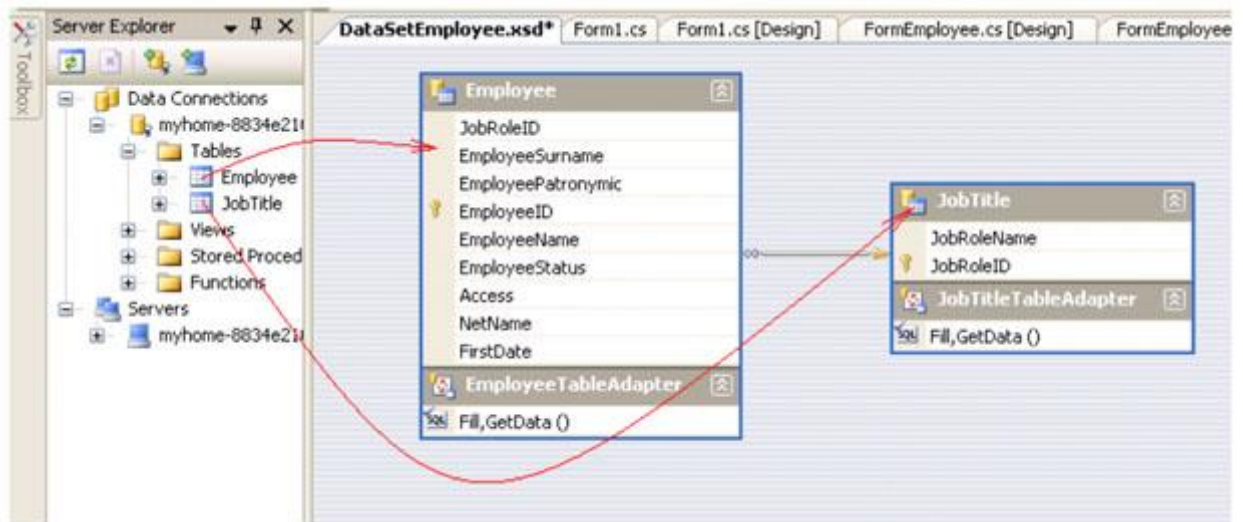
**Рис. 8.9.** Выбор нового компонента - DataSet

После нажатия кнопки *Add* система генерирует *класс DataSetEmployee*, который добавляется в решение проекта ([рисунок 8.10](#)).



**Рис. 8.10.** 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 8.11](#)).



[увеличить изображение](#)

**Рис. 8.11.** Добавление таблиц к DataSet

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

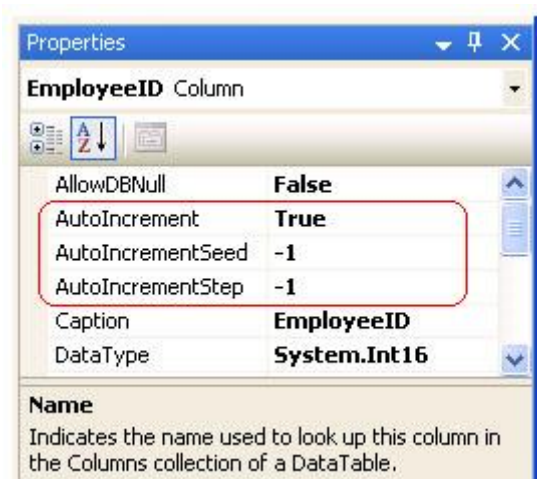
При формировании класса `DataSetEmployee` необходимо учесть то, что первичные ключи таблиц `Employee` и `JobTitle` являются суррогатными и автоматически формируются (*ключ* со свойством `AutoIncrement`) источником данных (например, *MS SQL Server*). При формировании новых записей в приложении необходимо обеспечить уникальность первичных ключей для таблиц объекта `DataSetEmployee`. 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 значения `AutoIncrementSeed` 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 `EmployeeID` таблицы `Employee` приведена на [рисунке 8.12](#).



**Рис. 8.12.** Установка свойств для колонки EmployeeID

Аналогичные установки свойств `AutoIncrement`, `AutoIncrementSeed` и `AutoIncrementStep` необходимо сделать и для колонки `JobTitleID` таблицы `JobTitle`.

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

```
DataSetEmployee dsEmployee = new DataSetEmployee();
DataSetEmployeeTableAdapters.EmployeeTableAdapter daEmployee =
    new Project_K4И_01.DataSetEmployeeTableAdapters.
EmployeeTableAdapter();
DataSetEmployeeTableAdapters.JobTitleTableAdapter daJobTitle =
    new Project_K4И_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 `Employee` создадим 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` и `dsEmployee.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 практическое занятие

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 заполнения таблиц `DataSet`.
6. Протестировать работу приложения.

## СПИСОК ЛИТЕРАТУРЫ

### Перечень основной литературы:

1. Иванова Г.С. Объектно-ориентированное программирование [Электронный ресурс]: учебник/ Иванова Г.С., Ничушкина Т.Н. — Электрон. текстовые данные. — Москва: Московский государственный технический университет имени Н.Э. Баумана, 2014. — 456 с. — Режим доступа: <http://www.iprbookshop.ru/94030.html>. — ЭБС «IPRbooks».
2. Маляров А.Н. Объектно-ориентированное программирование [Электронный ресурс]: учебник для технических вузов/ Маляров А.Н.— Электрон. текстовые данные. — Самара: Самарский государственный технический университет, ЭБС АСВ, 2017.— 332 с.— Режим доступа: <http://www.iprbookshop.ru/91772.html>.— ЭБС «IPRbooks».
3. Мурадханов С.Э. Информатика и программирование: объектно-ориентированное программирование (на основе языка С#) [Электронный ресурс]: учебник/ Мурадханов С.Э., Широков А.И. — Электрон. текстовые данные. — Москва: Издательский Дом МИСиС, 2015. — 309 с. — Режим доступа: <http://www.iprbookshop.ru/98855.html>.— ЭБС «IPRbooks».

### Перечень дополнительной литературы:

1. Зыков С.В. Введение в теорию программирования. Объектно-ориентированный подход [Электронный ресурс]: учебное пособие/ Зыков С.В.— Электрон. текстовые данные.— Москва: Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2021.— 187 с.— Режим доступа: <http://www.iprbookshop.ru/102007.html>.— ЭБС «IPRbooks».
2. Николаев Е.И. Объектно-ориентированное программирование [Электронный ресурс]: учебное пособие/ Николаев Е.И. — Электрон.текстовые данные. — Ставрополь: Северо-Кавказский федеральный университет, 2015. — 225 с. — Режим доступа: <http://www.iprbookshop.ru/62967.html>. — ЭБС «IPRbooks».
3. Сорокин А.А. Объектно-ориентированное программирование [Электронный ресурс]: учебное пособие. Курс лекций/ Сорокин А.А. — Электрон.текстовые данные. — Ставрополь: Северо-Кавказский федеральный университет, 2014. — 174 с. — Режим доступа: <http://www.iprbookshop.ru/63110.html>. — ЭБС «IPRbooks».