Часть II

РЕАЛИЗАЦИЯ И ЭКСПЛУАТАЦИЯ БАЗ ДАННЫХ

Раздел V

ПРОГРАММНЫЙ ИНТЕРФЕЙС ДОСТУПА К ДАННЫМ

Лекция 20

Технология ADO.NET

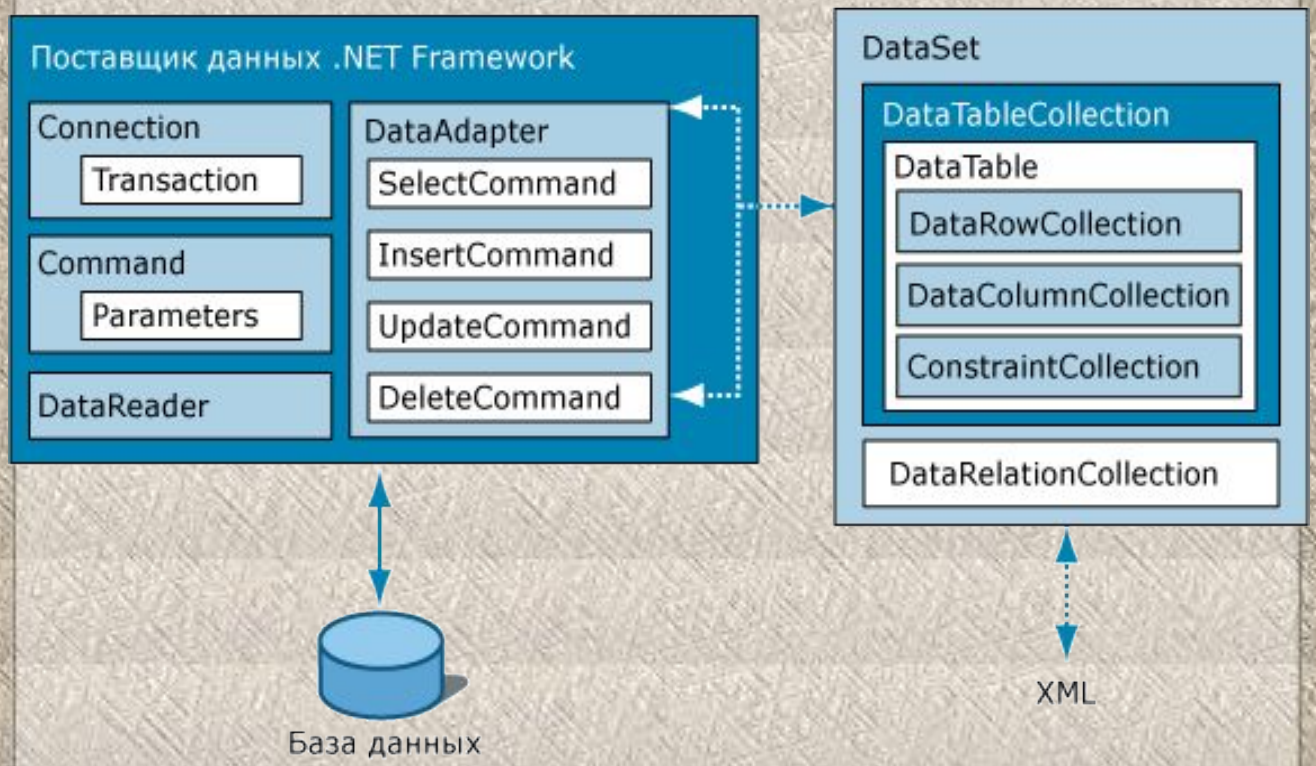
§1 Доступ к данным в ADO.NET

ADO.NET

ADO.NET – новый этап в технологии ActiveXDataObjects. Эта модель доступа к данным создана специально для использовании в Web-приложениях.

Если раньше в ADO упор делался на постоянное соединение с базой данных, то в технологии использования ADO.NET изначально заложена возможность работы приложения в состоянии «разрыва» соединения с базой данных.

Объектная модель ADO.NET



Объектная модель ADO.NET Уровень данных

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 и манипуляция данными на уровне исходных таблиц (выборка, сортировка, добавление, удаление, обновление).

Объектная модель ADO.NET Уровень управления

Уровень управления. 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**.

Если выполнялся процесс выборки информации из базы данных, для хранения результатов выборки используется объект **DataSet**.

Объектная модель ADO.NET Уровень приложения

Уровень приложения. Это набор объектов, позволяющих хранить и отображать данные на компьютере конечного пользователя.

Для хранения информации используется объект **DataSet**, а для отображения данных имеется довольно большой набор элементов управления (DataGrid, TextBox, ComboBox, Label и т.п.).

В Visual Studio .Net можно вести разработку двух типов приложений, это:

- традиционные Windows-приложения (на основе Windows-форм), которые реализованы в виде exe-файлов, запускаемых на компьютере пользователя;
- Web-приложения (на основе Web-форм), которые работают в оболочке браузера.

Для хранения данных на уровне обоих типов приложений используется объект **DataSet**.

Классы ADO.NET

ADO.NET – это библиотека .NET классов, которые позволяют подсоединяться к данным и манипулировать ими.

С целью разграничения функциональности классов **ADO.NET** они рассредоточены по различным пространствам имен.

В **ADO.NET** пространства имен используются для отделения различных частей модели управляемого поставщика данных.

Пространствам имен	Описание
System.Data	Пространство имен включает в себя общие структуры данных, не зависящие от конкретного поставщика. В него входит класс **DataSet** и целое семейство связанных с ним классов (**DataTable**, **DataColumn**, **DataRow**, DataRelation, Constraint и т.п.).
Поставщики данных	
System.Data.SqlClient	Управляемый поставщик SQLServer
System.Data.OleDb	Управляемый поставщик OLEDB
System.Data.Odbc	Управляемый поставщик ODBC

Структура данных ADO.NET

В **ADO.NET** есть два основных способа, обеспечивающих взаимодействие приложения с данными:

1. Использование набора данных (объект **DataSet**);

работа непосредственно с элементами базы данных: таблицами, представлениями, хранимыми процедурами и т.п. (объект **DataCommand**).

В «отсоединенной» модели работы с данными на основе **DataSet** разработчик создает в памяти компьютера некоторое пустое хранилище, загружает его данными, используя адаптер данных (объект **DataAdapter**), работает с этой информацией (сортирует, фильтрует, изменяет), затем, по мере необходимости, через тот же адаптер данных, возвращает все изменения в исходную базу данных.

Структура данных ADO.NET

2.В качестве альтернативы можно работать непосредственно с базой данных.

В этой модели используется объект **DataCommand**, в котором содержится SQL-запрос или имя хранимой процедуры. Команда запускается на выполнение, и если команда не возвращает результата (например, удаление записей), то все действия команды выполняются непосредственно над объектами базы данных (например, удаляется запись из таблицы). Если в результате работы команды из базы данных возвращается набор записей, то используется объект **DataReader** для выборки данных.

В некоторых случаях задача вообще не решается путем использования набора данных. Например, если требуется создать объект базы данных (типа таблица), то это можно сделать только с помощью команд (объектов **DataCommand**).

§2 Соединение с источником данных

Соединение с источником данных

Для перемещения данных между их постоянным хранилищем и приложением в первую очередь необходимо создать соединение с источником данных (Connection).
В арсенале **ADO.NET** для этих целей имеется ряд объектов:

Объект	Описание
SQLConnection	Объект, позволяющий создать соединение с базами данных MSSQLServer
OleDbConnection	Объект, позволяющий создать соединение с любым источником данных (простые текстовые файлы, электронные таблицы, базы данных) через OLEDB
OdbcConnection	объект, позволяющий создать соединение с ODBC-источниками данных.

Жизненный цикл объекта Connection

Жизненный цикл объекта **Connection** состоит из таких этапов как:

- объявление объекта соединения;
- создание объекта соединения;
- определение строки соединения;
- использование соединения, например, для создания команды;
- открытие соединения;
- выполнение команды;
- закрытие соединения.

Объявление объекта соединения

```
publicclass Form1 : System.Windows.Forms.Form
{
     private System.Data.SqlClient.SqlConnection sqlConnection1;
     private System.Data.OleDb.OleDbConnection oleDbConnection1;
     private System.Data.Odbc.OdbcConnection odbcConnection1;
     private System.Data.Odbc.OdbcConnection odbcConnection2;
...
}
```

Создание соединения

(Операторы создания объектов соединения помещаются в блок инициализации)

```
privatevoid InitializeComponent()
{
     this.sqlConnection1 = new System.Data.SqlClient.SqlConnection();
     this.oleDbConnection1 = new System.Data.OleDb.OleDbConnection();
     this.odbcConnection1 = new System.Data.Odbc.OdbcConnection();
     this.odbcConnection2 = new System.Data.Odbc.OdbcConnection();
…
}
```

Строка соединения

Первое свойство объекта соединения, которое необходимо определить в блоке инициализации для установления связи с базой данных – это строка соединения **ConnectionString**.
В строке соединения управляемых поставщиков необходимо, как минимум, указать:

- местоположение базы данных
- требуемую аутентификационную информацию.

Помимо этого, каждый поставщик данных определяет дополнительные параметры соединения. Если в строке соединения не указаны значения всех возможных параметров, они считаются установленными по умолчанию.

Строка соединения управляемого поставщика SQL Server

Строки соединения управляемого поставщика SQL Server содержат множество параметров. Самыми распространенными из них являются:

Параметров	Описание
DataSource	Имя сервера баз данных
InitialCatalog	База данных, находящаяся на сервере
UserID	Идентификатор пользователя, который должен применяться для аутентификации пользователя на сервере баз данных
PWD	Пароль, который должен применяться для аутентификации пользователя на сервере баз данных
IntegratedSecurity	Авторизация Windows, если этот параметр содержит True, в противном случае ввод имени и пароля

Пример 1
строки соединения управляемого поставщика SQL Server

Например, строка соединения с базой данных **Students**, расположенной на MS SQL Server с именем **(local)** с Windows-авторизацией будет выглядеть следующим образом:

```
sqlConnection1.ConnectionString = "data source=\"(local)\";initial
catalog=\" Students\"; IntegratedSecurity=true";
```

Строки соединения управляемого поставщика OLE DB

Строки соединения управляемого поставщика **OLE DB** похожи на строки соединения SQL Server.

Все параметры строки соединения, за исключением параметра **Provider** (Поставщик), определяются специфическим поставщиком **OLEDB**.

В качестве значений параметра **Provider** могут быть

Провайдер	Описание
SQLOLEDB.1	Для SQL Server
Microsoft.Jet.OLEDB.4.0	Для Microsoft Access
PostgreSQL	Для базы данных PostgreSQL

Пример 2
Строки соединения управляемого поставщика OLE DB

Например, строка соединения с базой данных **Students**, расположенной на MS SQL Server с именем **(local)** с Windows-авторизацией будет выглядеть следующим образом и управлением поставщика OLE DB:

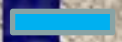
```
oleDbConnection1.ConnectionString = "data source=\"(local)\";initial
catalog=\"Students\"; IntegratedSecurity=true"; Provider=
"SQLOLEDB.1";
```

Строка соединения управляемого поставщика ODBC

Строки соединения управляемого поставщика ODBC немного отличаются от строк соединения SQL Server или OLE DB.

Управляемый поставщик ODBC поддерживает два различных метода создания строки соединения:

- Создание строки соединения на основе имени источника данных (**DataSourceNameDSN**);
- Использование динамических строк соединения.



Использование **DSN** заключается в том, что каждый компьютер должен либо быть специально настроенным, либо иметь доступ к **DSN**-файлам.

Использование **DSN** позволяет сохранить определенный контроль над строками соединения. Так, если меняется местоположение сервера или аутентификационная информация, разработчику придется изменить всего лишь атрибуты **DSN**, а не программный код.

Атрибуты **DSN** можно использовать также для динамического генерирования информации о соединении. В этом случае параметры строки соединения, такие как **DRIVER** и **SERVER**, можно указать непосредственно в строке соединения, а не прятать их в атрибуте **DSN**.

Пример 3
Строки соединения управляемого поставщика ODBC

Например, строка соединения на основе имени источника данных может выглядеть так:

```
odbcConnection1.ConnectionString = "UID=usera; DSN=stud; PWD=123";
```

А динамическая строка соединения с тем же источником данных выглядит следующим образом:

```
odbcConnection2.ConnectionString = "UID=usera; DRIVER=SQL Server;
PWD=123; SERVER=ITS-SERVER";
```

Обработка ошибок в .NET

Для перехвата исключений и их обработки в среде .NET предусмотрена структура ***try-catch-finally***:

```
try
{
     // попытка соединения с БД
     …
     MessageBox.Show("Успешное соединение SqlServer!");
}
catch(Exception ex)
{
     …
     // если исключение было передано этому фрагменту кода,
     // значит, возникло исключение, о чем необходимо сообщить
     // пользователю
     MessageBox.Show("Соединения нет!"+ex.Message);
}
finally
{
     …
     // этот фрагмент кода выполняется вне зависимости от того,
     // было сгенерировано исключение или нет
}
```

При возникновении ошибки платформа .NET генерирует исключение.

Смысл обработки исключений заключается в том, что при неудачной попытке что-то выполнить процесс генерирует объект исключения, который может быть перехвачен вызывающим кодом.

Открытие и закрытие соединения

Объекты **Connection** имеют два базовых метода для открытия и закрытия соединения (**Open** и **Close**)

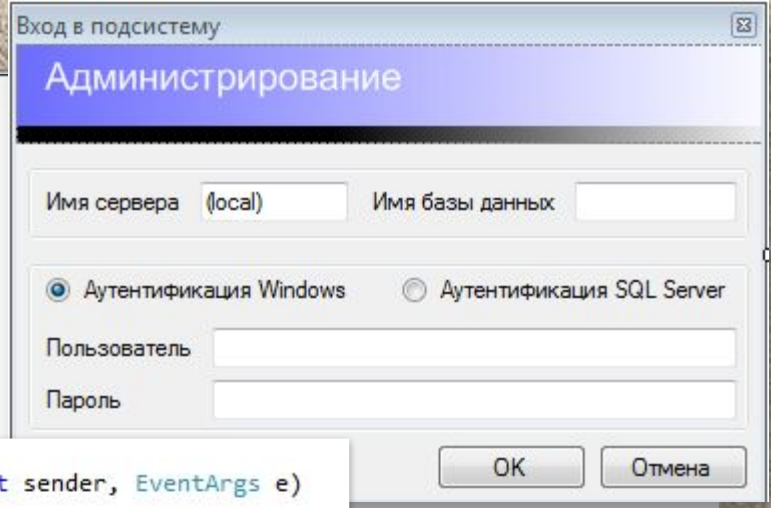
```
privatevoid Form1_Load(object sender, System.EventArgs e)
{
//методы открытия и закрытия объекта  sqlConnection
     try
     {
          this.sqlConnection1.Open();
          MessageBox.Show("Успешное SQL соединение");
          this.sqlConnection1.Close();
     }
     catch(Exception ex)
     {
          MessageBox.Show("Нет SQL соединения"+ex.Message);
     }
//методы открытия и закрытия объекта  oleDbConnection
     try
     {
          this.oleDbConnection1.Open();
          MessageBox.Show("Успешное OLE DB соединение");
          this.oleDbConnection1.Close();
     }
     catch(Exception ex)
     {
          MessageBox.Show("Нет OLE DB соединения"+ex.Message);
     }
//методы открытия и закрытия объекта  odbcConnection1
     try
     {
          this.odbcConnection1.Open();
          MessageBox.Show("Успешное ODBC1 соединение");
          this.odbcConnection1.Close();
     }
     catch(Exception ex)
     {
          MessageBox.Show("Нет ODBC1 соединения"+ex.Message);
     }
}
```

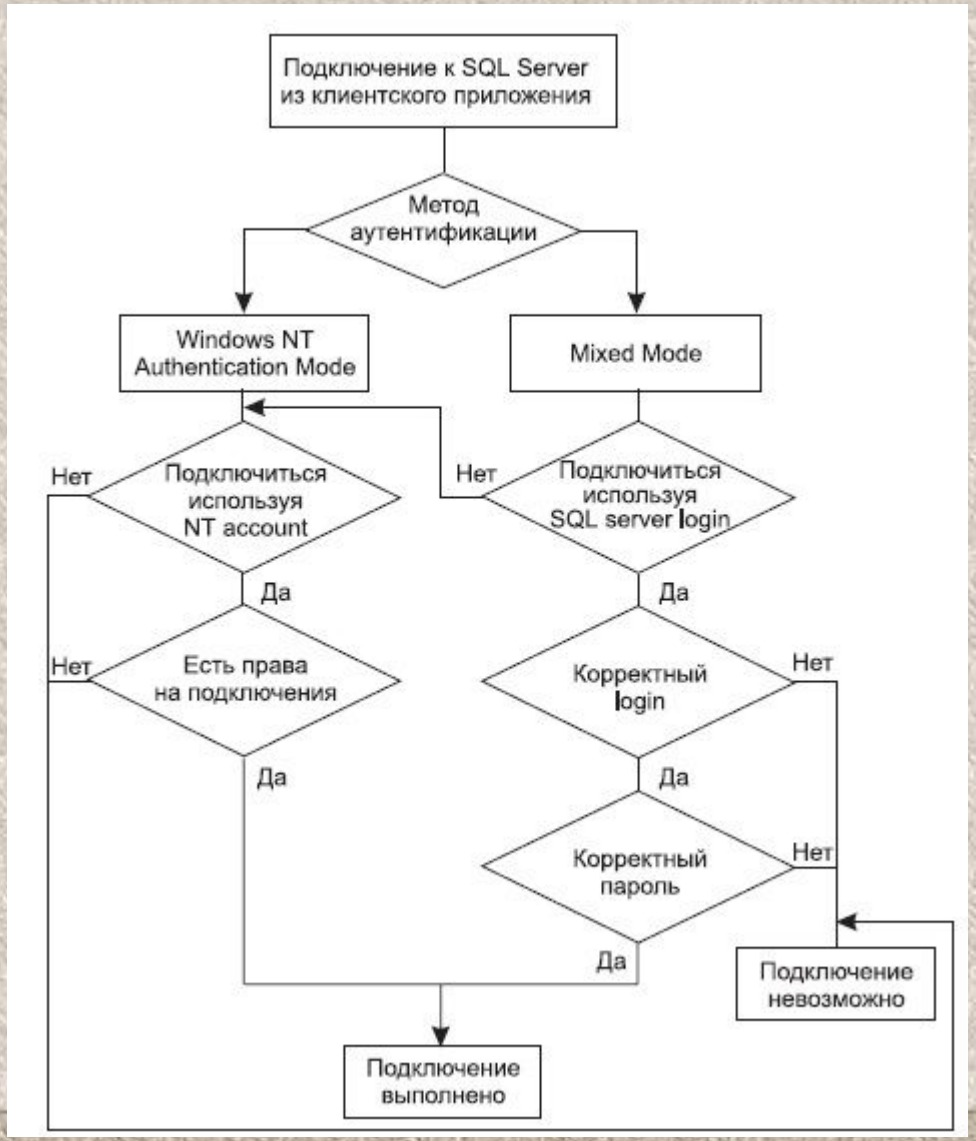
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace AdministrationApplication
{
    public partial class AutentificationForm : Form
    {
        public string servername;
        public string databasename;
        public string login;
        public string pass;
        public bool IntSeq = true;
        public bool OKPressed;


        public AutentificationForm()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            if ((textBox1.Text == "") || (textBox2.Text == "") ||  ((radioButton2.Checked == true) && (textBox3.Text == "")) )
                MessageBox.Show("Введите данные для подключения", "Подключение к серверу", MessageBoxButtons.OK, MessageBoxIcon.Warning);
            else
            {
                servername = textBox1.Text;
                databasename = textBox2.Text;
                login = textBox3.Text;
                pass = textBox4.Text;
                OKPressed = true;
                IntSeq = radioButton1.Checked;
                this.Close();
            }
        }
```

```
private void button2_Click(object sender, EventArgs e)
{
    OKPressed = false;
    this.Close();
}
```

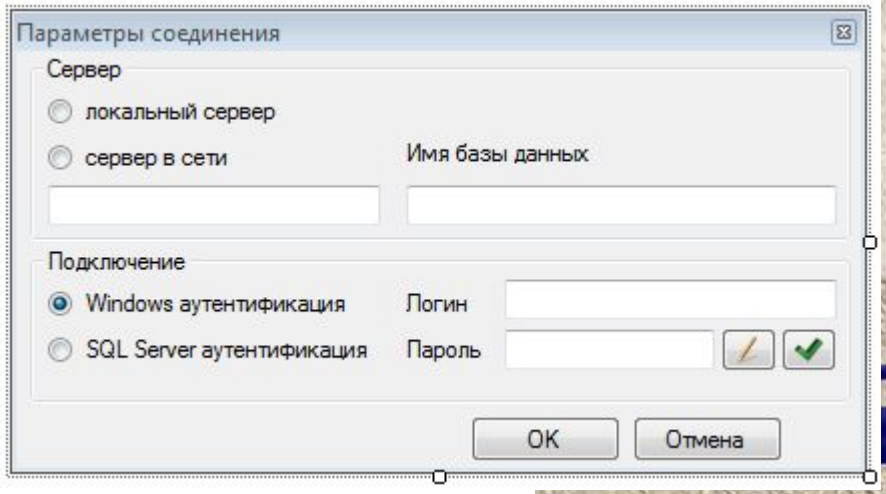
```
private void radioButton1_CheckedChanged(object sender, EventArgs e)
{
   label3.Enabled = label4.Enabled =
   textBox3.Enabled = textBox4.Enabled = !(radioButton1.Checked);

}
```

```
private void radioButton2_CheckedChanged(object sender, EventArgs e)
{
    label3.Enabled = label4.Enabled =
    textBox3.Enabled = textBox4.Enabled = (radioButton2.Checked);
}
```




Подключение к SQL Server
из клиентского приложения
Метод
аутентификации
Windows NT
Authentication Mode
Mixed Mode
Нет
Подключиться
используя
NT account
Нет
Подключиться
используя
SQL server login
Да
Да
Нет
Есть права
на подключения
Нет
Корректный
login
Да
Да
Корректный
пароль
Нет
Да
Подключение
невозможно
Подключение
выполнено

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Xml;
using System.Security.Cryptography;

namespace Main
{
    public partial class ConnectionSettingsForm : Form
    {
        string srv = "";
        string icat = "";
        string iseq = "";
        string username = "";
        string passhash = "";
        string HashString = "";
        bool IsEditing = false;

        public ConnectionSettingsForm()
        {
            InitializeComponent();
            System.Data.SqlClient.SqlConnectionStringBuilder scsb = new System.Data.SqlClient.SqlConnectionStringBuilder();
            XmlTextReader xm = new XmlTextReader("Settings.xml");
            xm.ReadStartElement();
            while (xm.Read())
            {
                if (xm.Name == "server") srv = xm.ReadElementContentAsString();
                if (xm.Name == "initial") icat = xm.ReadElementContentAsString();
                if (xm.Name == "IntegratedSecurity") iseq = xm.ReadElementContentAsString();
                if (xm.Name == "Username") username = xm.ReadElementContentAsString();
                if (xm.Name == "Password") passhash = xm.ReadElementContentAsString();
            }
            xm.Close();

            if (srv == "(local)")
                radioButton1.Checked = true;
            else
            {
                radioButton2.Checked = true;
                textBox1.Text = srv;
            }
            textBox2.Text = icat;
            if (iseq == "True")
                radioButton3.Checked = true;
            else
            {
                radioButton4.Checked = true;
                textBox3.Text = username;
                textBox4.Text = passhash;
            }

        }
```



```
private void button1_Click(object sender, EventArgs e)
{
    if ((textBox1.Text == "") || (textBox1.Text == ""))
        MessageBox.Show("Необходимо указать все значения!", "Параметры подключения", MessageBoxButtons.OK, MessageBoxIcon.Asterisk);
    else
    {
        XmlTextWriter xml = new XmlTextWriter("Settings.xml",
            System.Text.Encoding.Unicode);
        xml.Formatting = Formatting.Indented;
        xml.WriteStartDocument();
        xml.WriteStartElement("Settings");
        if (radioButton1.Checked == true)
            srv = "(local)";
        else
            srv = textBox1.Text;
        icat = textBox2.Text;
        xml.WriteStartElement("server");
        xml.WriteString(srv);
        xml.WriteEndElement();

        xml.WriteStartElement("initial");
        xml.WriteString(icat);
        xml.WriteEndElement();

        if (radioButton3.Checked == true)
        {
            iseq = "True";
            username = "";
            passhash = "";
        }
        else
        {
            iseq = "False";
            username = textBox3.Text;
            passhash = textBox4.Text;
        }
        xml.WriteStartElement("IntegratedSecurity");
        xml.WriteString(iseq);
        xml.WriteEndElement();

        xml.WriteStartElement("Username");
        xml.WriteString(username);
        xml.WriteEndElement();

        xml.WriteStartElement("Password");
        xml.WriteString(passhash);
        xml.WriteEndElement();

        xml.WriteEndElement();
        xml.WriteEndDocument();
        xml.Close();
        this.Close();
    }
}
```

```
private void radioButton1_CheckedChanged(object sender, EventArgs e)
{
    if (radioButton1.Checked == true)
    {
        textBox1.Text = "(local)";
        textBox1.Enabled = false;
    }
}
```

```
private void radioButton2_CheckedChanged(object sender, EventArgs e)
{
    if (radioButton2.Checked == true)
    {
        textBox1.Text = "";
        textBox1.Enabled = true;
    }
}
```

```
private void button2_Click(object sender, EventArgs e)
{
    this.Close();
}
```

```
private void radioButton3_CheckedChanged(object sender, EventArgs e)
{
    if (IsEditing == false)
        label1.Enabled = label2.Enabled = textBox3.Enabled =
            button3.Enabled = !(radioButton3.Checked);
    else
    {
        label1.Enabled = label2.Enabled = textBox3.Enabled = textBox4.Enabled =
button3.Enabled = button4.Enabled =  !(radioButton3.Checked);
        textBox4.Text = HashString;
        IsEditing = false;
    }
}
```

```
private void radioButton4_CheckedChanged(object sender, EventArgs e)
{
    label1.Enabled  = label2.Enabled = textBox3.Enabled =
        button3.Enabled = radioButton4.Checked;
}
```

```
private void button3_Click(object sender, EventArgs e)
{
    IsEditing = true;
    HashString = textBox4.Text;
    textBox4.Text = "";
    button3.Enabled = false;
    textBox4.Enabled = true;
    button4.Enabled = true;

}
```

```
    private void button4_Click(object sender, EventArgs e)
    {
        IsEditing = false;
        textBox4.Text = getMd5Hash(textBox4.Text);
        button4.Enabled = false;
        textBox4.Enabled = false;
        button3.Enabled = true;
    }

    static string getMd5Hash(string input)
    {
        MD5 md5Hasher = MD5.Create();
        byte[] data = md5Hasher.ComputeHash(Encoding.Default.GetBytes(input));
        StringBuilder sBuilder = new StringBuilder();
        for (int i = 0; i < data.Length; i++)
        sBuilder.Append(data[i].ToString("x2"));
        return sBuilder.ToString();
    }


}
```

§2 Работа с объектом DataCommand

Команда данных

Для выполнения основных задач, связанных с взаимодействием с базами данных, можно использовать объекты команд.

Команда данных содержит ссылку на SQL-запрос или хранимую процедуру, которые собственно и реализуют конкретные действия.

Команда данных – это экземпляр класса:

`System.Data.Odbc.OdbcCommand`

или

`System.Data.OleDb.OleDbCommand`

или

`System.Data.SqlClient.SqlCommand.`

Действия объекта DataCommand

С использованием объекта **DataCommand** в приложении можно выполнять следующие действия:

1. Исполнять команды **SELECT**, которые возвращают набор записей. Причем результат выборки можно обрабатывать непосредственно, без его загрузки в набор данных ***DataSet***.
2. Выполнять команды, обеспечивающие создание, редактирование, удаление объектов базы данных (например, таблиц, хранимых процедур и т.п.).
3. Выполнять команды, обеспечивающие получение информации из баз данных в виде хранимых процедур.

Действия объекта DataCommand

4. Выполнять динамические SQL-команды, позволяющие модифицировать, вставлять или удалять записи непосредственно в базе данных, вместо того, чтобы редактировать таблицы набора данных **DataSet**, а затем копировать эти изменения в базу данных.
5. Выполнять команды, которые возвращают скалярное, то есть единственное значение.
6. Выполнять команды, которые возвращают данные из базы данных SQL Server в формате **XML**. (Эта возможность используется в Интернет-приложениях.)
   Например, когда нужно выполнить запрос и получить данные в формате XML, чтобы преобразовать данные к HTML-формату и затем отправить их браузеру.

Создание объекта DataCommand

Существует два основных способа создания объекта **DataCommand**.

```
System.Data.Odbc.OdbcConnection con1;// соединение
System.Data.Odbc.OdbcCommand cmd1;//команда
…
cmd1=new System.Data.Odbc.OdbcCommand();

System.Data.Odbc.OdbcConnection con1;// соединение
System.Data.Odbc.OdbcCommand cmd2; //команда
…
cmd2=con1.CreateCommand();
```

Типы команд

Команда – мощный инструмент, позволяющий проводить сложные операции с базой данных.
В **ADO.NET** существует три типа команд:

Text – текстовая команда состоит из SQL-инструкций, указывающих управляемому поставщику на необходимость выполнения определенных действий на уровне базы данных. Текстовые команды передаются в базу данных без предварительной обработки, за исключением случаев передачи параметров;

StoredProcedure – хранимая процедура; эта команда вызывает процедуру, которая хранится в самой базе данных;

TableDirect – команда такого типа предназначена для извлечения из базы данных полной таблицы.

Типы команд

Тип команды устанавливается в свойстве **CommandType**, которое по умолчанию имеет значение **Text**, а сам текст команды прописывается в свойстве **CommandText**.

```
cmd1=new System.Data.SqlClient.SqlCommand ();
cmd1.Connection=con1;
cmd1.CommandText="DELETE FROM Студенты WHERE Код_студента=1";
```

Методы выполнения команд

За подготовкой команды следует ее выполнение. В **ADO.NET** существует несколько способов выполнения команд, которые отличаются лишь информацией, возвращаемой из базы данных.

Рассмотрим методы выполнения команд, поддерживаемые всеми управляемыми поставщиками:

ExecuteNonQuery() – этот метод применяется для выполнения команд, которые не должны возвращать результирующий набор данных. Так как при вызове данного метода возвращается число строк, добавленных, измененных или удаленных в результате выполнения команды, он может использоваться в качестве индикатора успешного выполнения команды;

Методы выполнения команд

ExecuteScalar() – этот метод выполняет команду и возвращает первый столбец первой строки первого результирующего набора данных. Данный метод может быть полезен для извлечения из базы данных итоговой информации; количества, максимального или минимального значения, итоговой суммы или среднего значения;

ExecuteReader() – этот метод выполняет команду и возвращает объект **DataReader**, представляющий собой однонаправленный поток записей базы данных.

```
System.Data.SqlClient.SqlConnection con1;// соединение
System.Data.SqlClient.SqlCommand cmd1;//команда
System.Data.SqlClient.SqlCommand cmd2; //команда
…
cmd1=new System.Data.SqlClient.SqlCommand ();
cmd1.Connection=con1;
cmd1.CommandText="DELETE FROM Студенты WHERE Код_студента=1";
try
{
      cmd1.ExecuteNonQuery();
      MessageBox.Show("Успешное удаление!");
}
catch(Exception ex)
{
     MessageBox.Show("Удаление не удалось!"+ex.Message);
}

cmd1.CommandText="UPDATE Студенты SET стипенд=стипенд*10
WHERE ном_сту=375";
try
{
     cmd1.ExecuteNonQuery();
     MessageBox.Show("Успешное изменение!");
}
catch(Exception ex)
{
     MessageBox.Show("Измененииене удалось!"+ex.Message);
}
cmd1.CommandText="INSERT INTO Студенты VALUES(2,'Николаев',
'08-ИС',400000,'16.01.2008')";
try
{
     cmd1.ExecuteNonQuery();
     MessageBox.Show("Успешная вставка!");
}
catch(Exception ex)
{
     MessageBox.Show("Вставка не удалась!"+ex.Message);
}
cmd1.Dispose();
```

Использование метода **ExecuteNonQuery()**

Использование метода **ExecuteScalar()**

```
System.Data.Odbc.OdbcCommand cmd=
new System.Data.Odbc.OdbcCommand("SELECT MAX(ср_балл) FROM
Успеваемость",con1);
…
int result= (int)cmd.ExecuteScalar();
```

Пример вызова хранимой процедуры “MarkaNeisp”

Если в приложении используется объект **DataCommand**, который работает непосредственно с элементами базы данных, то выполняемые SQL-запросы и хранимые процедуры обычно требуют параметров.

Перед выполнением таких запросов необходимо определить значения параметров.
Объект **DataCommand** поддерживает коллекцию **Parameters**. Прежде чем выполнить команду, необходимо установить значение для каждого параметра команды.

```
DataTable FilTable = newDataTable();
SqlCommand cmd = newSqlCommand("MarkaNeisp", cn);
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.Add(newSqlParameter("@Mar", SqlDbType.Char));
cmd.Parameters["@Mar"].Value = TextBox1.Text;
SqlDataAdapter FilAdap = newSqlDataAdapter(cmd);
FilAdap.Fill(FilTable);
dataGridView1.DataSource = FilTable;
```

§3 Отсоединенный набор данных DataSet

Объект DataSet

Объект **DataSet** – это:

- набор информации, извлеченной из базы данных; доступ к этому набору осуществляется в отсоединенном режиме;
- база данных, расположенная в памяти;
- сложная реляционная структура данных со встроенной поддержкой XML-сериализации.

Роль объекта **DataSet** в **ADO.NET** заключается в предоставлении отсоединенного хранилища информации, извлеченной из базы данных, и в обеспечении для .NET возможностей базы данных, хранимой в памяти.

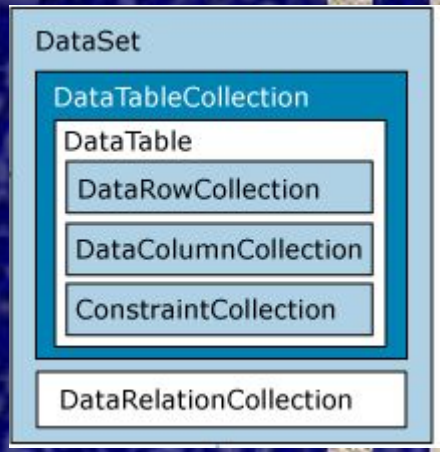
Объект **DataSet** – это коллекция структур данных, использующихся для организации отсоединенного хранилища информации.

Так как объект **DataSet** не связан с базой данных, его можно воспринимать как реляционное хранилище данных.

Объект DataSet

Объект **DataSet** состоит из нескольких связанных друг с другом структур данных. Концептуально он представляет собой полный набор реляционной информации. Внутри объекта **Dataset** могут храниться пять объектов:

- **DataTable** - набор данных, организованный в столбцы и строки;
- **DataRow**– коллекция данных, которая представляет собой одну строку таблицы *DataTable*, объект *DataRow* является фактическим хранилищем данных;
- **DataColumn** – коллекция правил, описывающая данные, которые можно хранить в объектах *DataRow*;
- **Constraint** – данный объект используется для определения бизнес – правил объекта *DataTable* и задает допустимость хранения определенных данных в объекте *DataTable*;
- **DataRelation** –описание связей между объектами *DataTable*.



Работа с объектом DataSet

Данные, которые хранятся внутри объекта **DataSet**, содержат не только информацию, необходимую для поддержки отсоединенного кэша базы данных, но также предоставляют возможность перемещаться по нему как по некоторой иерархической структуре.

Основным предназначением объекта **DataSet** является хранение и изменение данных.

Объекты **DataRow** являются основным хранилищем данных внутри объекта **DataSet**. Объект **DataRow** содержит массив значений, представляющих собой строку объекта **DataTable**. Объекты **DataRow** доступны из объекта **DataTable** через свойство **Rows**.

Выборка строки

Пусть имеется набор данных с именем *ds1* из таблицы «Автомобиль»

Пример выборки данных из объекта **DataSet** по номеру строки в текстовые поля:

```
int n=10;
DataRow r=this.ds1.Tables["Автомобиль"].Rows[n];
textBox1.Text=r["Ид_авто"].ToString();
textBox2.Text=r["Марка"].ToString();
textBox3.Text=r["Модель"].ToString();
textBox4.Text=r["Цвет"].ToString();
textBox5.Text=r["Год_выпуска"].ToString();
```

Добавление строки

Для создания новой строки можно использовать соответствующие методы (**NewRow()** и **Add()**) объекта **DataTable**.

Следует отметить, что метод **NewRow()** сам по себе не добавляет строку в объект **DataTable**. Для этого необходимо вызвать метод **Add()**, передав ему в качестве параметра объект строки.

Пример добавления строки в объект **DataSet**:

```
DataRow r=ds1.Tables["Автомобиль"].NewRow();
     r["Ид_авто"]=1;
     r["Марка "]="BMW";
     r["Модель "]="X5";
     r["Цвет "]="Black";
     r["Год_выпуска "]="2007";
ds1.Tables["Автомобиль"].Rows.Add(r);
```

Удаление строки

При использовании отсоединенных данных к удалению строки из коллекции предъявляется особое требование: строка должна продолжать существовать до тех пор, пока хранилище данных не будет обновлено с помощью объекта **DataSet**. Удаление строки может быть осуществлено, например, с помощью метода **Delete()** объекта **DataRow**. В этом случае строка удаляет себя сама.

Пример удаления строки :

```
int n=10;
DataRow r=this.ds1.Tables["Автомобиль "].Rows[n];
r.Delete();
```

Изменение строки

Индексаторы класса **DataRow** позволяют установить новые значения столбцов строки, например:

```
r["фамилия"]="Иванов";
```

Однако при определении нового значения столбца объект **DataRow** сгенерирует исключение в том случае, если это значение будет конфликтовать со свойством **DataType** объекта **DataColumn**.

Существуют ситуации, в которых изменения в конкретную строку **DataRow** необходимо вносить параллельно. Обычно это делается тогда, когда одно изменение приводит к нарушению некоторого ограничения или когда необходимо иметь возможность отмены изменений перед их внесением в базу данных.
В этом случае используются методы **BeginEdit(), EndEdit()** и **CancelEdit()** класса **DataRow**.

Изменение строки

Как только будет вызван метод **BeginEdit(),** изменения перестанут отражаться на объекте **DataRow** до тех пор, пока не будет вызван метод **EndEdit().** Если внесенные изменения окажутся ошибочными, можно вызвать метод **CancelEdit(),** который вернет строку в первоначальное состояние (состояние, в котором она находилась до вызова метода **BeginEdit()**).

Пример изменения строки:

```
//изменить запись
int n=10;
DataRow r=this.ds1.Tables["Клиент"].Rows[n];
r.BeginEdit();
r["фамилия"]="Иванов";
r["имя"]="Иван";
r["отчество"]="Иванович";
r["дата рождения"]="1.01.85";
r.EndEdit();
```

§4 Объект DataAdapter

DataAdapter

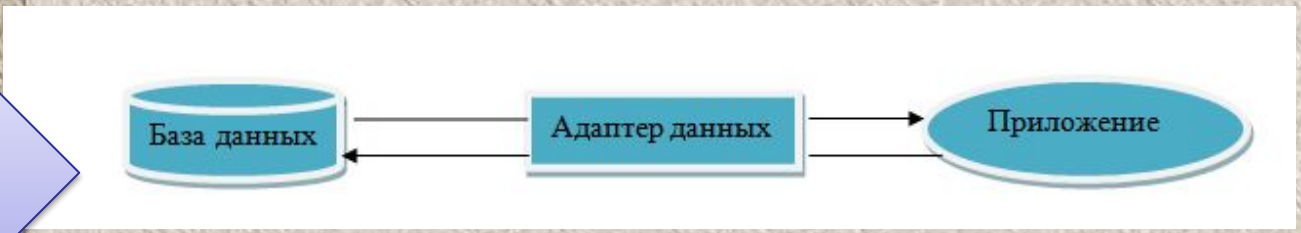
Объект DataAdapter

DataAdapter– один из важнейших объектов **ADO.NET**. Этот объект является посредником между источником данных и набором данных **DataSet**.

В приложениях **DataAdapter** обеспечивает считывание информации из базы данных и пересылку ее в **DataSet**, возврат изменений, сделанных пользователем, в исходную базу данных.

Задача модификации данных решается через использование команд на основе SQL-запросов и хранимых процедур.

Схема функционирования объекта **DataAdapter**



Типы адаптеров

В **ADO.NET** имеется несколько типов адаптеров данных:

Адаптер данных	Описание
System.Data.Odbc.OdbcDataAdapter	используется для работы с ODBC – источниками данных
System.Data.OleDb.OleDbDataAdapter	используется для работы с любым источником данных, доступным через OLEDB – провайдера
System.Data.SqlClient.SqlDataAdapter	используется для работы с данными, хранящимися в SQL Server

Объекты DataAdapter

Каждый объект **DataAdapter** обеспечивает обмен данными между одной таблицей источника данных (базы данных) и одним объектом **DataTable** в наборе данных **DataSet**.

Если **DataSet** содержит несколько таблиц (объектов **DataTable**), то необходимо иметь и несколько адаптеров данных.

Когда требуется заполнить данными таблицу в **DataSet**, вызывается соответствующий метод (**Fill**) объекта **DataAdapter**, который выполняет SQL-запрос или хранимую процедуру.

Точно также, когда необходимо модифицировать базу данных, вызывается соответствующий метод (**Update**) объекта **DataAdapter**, который вызывает на исполнение соответствующий SQL-запрос или хранимую процедуру.
В результате этого все изменения, внесенные пользователем в таблицы набора данных, будут возвращены в соответствующие таблицы базы данных.

Использование объекта DataAdapter для заполнения объекта DataSet данными

Объект **DataAdapter** является связующим звеном между объектом **DataSet** и хранилищем данных.

С помощью объекта **DataAdapter** можно заполнить объект **DataSet** информацией из хранилища данных, а также обновлять хранилище данных на основе объекта **DataSet**.

Объект **DataAdapter** состоит из четырех объектов **Command**, каждый из которых выполняет определенную задачу по модификации данных в базе данных:

- **SelectCommand,**
- **InsertCommand,**
- **UpdateCommand**
- **DeleteCommand.**

Основное предназначение объекта DataAdapter при заполнении объекта DataSet

Объект **DataAdapter** используется каждый раз, когда объект **DataSet** нуждается в непосредственном взаимодействии с источником данных.

Один из наиболее важных моментов заключается в том, что объект **DataAdapter** содержит объекты **Command** для каждой из основных операций, производимых над базами данных.

Основное предназначение объекта **DataAdapter** заключается в формировании «моста» между объектом **DataSet** и базой данных.

Заполнение объекта DataSet

Еще одной важной задачей объекта **DataAdapter** является минимизация времени, в течение которого соединение будет оставаться открытым.

При использовании объекта **DataAdapter** явного открытия или закрытия соединения не происходит.

DataAdapter знает, что соединение должно быть как можно более коротким, и самостоятельно управляет его открытием и закрытием.

Если использовать объект **DataAdapter** совместно с уже открытым соединением, то состояние соединения будет сохранено.

Заполнение объекта DataSet

Для заполнения **DataSet** информацией из базы данных необходимо:

1. Создать экземпляр класса **DataAdapter**, который является специализированным классом, выполняющим функцию контейнера для команд, осуществляющих чтение и запись информации в базу данных;
2. Создать экземпляр класса **DataSet**; только что созданный объект **DataSet** является пустым;
3. Вызвать метод **Fill()** объекта **DataAdapter** для заполнения объекта **DataSet** данными из запроса, определенного в объекте **Command**.

Пример заполнения набора данных Sto

```
…
DataSet Sto = newDataSet("Sto");
SqlDataAdapter AvtoAdap;
AvtoAdap = newSqlDataAdapter("SELECT * FROM Автомобиль", cn);
AvtoAdap.Fill(Sto, "Автомобиль");//заполнение данными набора данных
…
AvtoAdap.Update(Sto.Tables["Автомобиль"]);//сохранение изменений в БД
```

§5 Объект CommandBuilder

Класс CommandBuilder

Класс **CommandBuilder** отвечает за генерацию запросов по мере возникновения необходимости в них в объекте **DataAdapter**.

Каждый управляемый поставщик содержит собственную реализацию класса **CommandBuilder** *(***SqlCommandBuilder, OleDbCommandBuilder, OdbcCommandBuilder***).*

После создания объекта **CommandBuilder** его необходимо передать в конструктор объекта **DataAdapter**. Как только объект **CommandBuilder** узнает об объекте **DataAdapter**, он использует свойство **DataAdapter**.

SelectCommand, чтобы получить информацию о столбцах объекта **DataTable** и иметь возможность создать команды вставки, обновления и удаления данных.

Функционирование объекта CommandBuilder

Для того, чтобы гарантировать нормальное функционирование объекта **CommandBuilder**, необходимо учесть несколько моментов.

Свойство **DataAdapter.SelectCommand** должно содержать действительную команду, которая использовалась для заполнения обновляемого объекта **DataTable**.

Объект **CommandBuilder** применяет SQL- оператор SELECT для того, чтобы иметь возможность создавать операторы вставки, обновления и удаления данных.

Таблица **DataTable**, которая будет обновляться, должна либо содержать столбец уникальных значений, либо для нее должен быть определен первичный ключ.

Пример использования объекта **CommandBuilder**:

```
…
DataSet Sto = newDataSet("Sto");
SqlDataAdapter AvtoAdap;
SqlCommandBuilder AvtoBild;
AvtoAdap = newSqlDataAdapter("SELECT * FROM Автомобиль", cn);
AvtoBild = newSqlCommandBuilder(AvtoAdap);
AvtoAdap.Fill(Sto, "Автомобиль");//заполнение данными набора данных
…
AvtoAdap.Update(Sto.Tables["Автомобиль"]);//сохранение изменений в БД
```

§6 Доступ к данным в Windows-формах

Привязка данных в ADO.NET

Windows-формы поддерживают два типа привязки данных.

- Для элементов управления, содержащих единственное значение (таких, как **TextBox**, **CheckBox** и т.п.), Windows–формы поддерживают простую привязку данных,
- Для элементов управления, содержащих несколько значений (таких, как **ListBox**, **ComboBox**, **DataGrid** и т.п.) – сложную привязку данных.

Далее рассмотрим некоторые из них.

Сложная привязка данных к элементам управления

Для привязки данных к **ComboBox** необходимо определить значения трех его свойств:

Методы объектаComboBox	Описание
DataSource	экземпляр класса, реализующего интерфейс **List** (например, класс **DataTable**)
DisplayMember	свойство объекта – источника (**DataSource**), которое будет отображаться в элементе управления
ValueMember	идентификатор, хранящийся в элементе управления и определяющий строку объекта – источника, к которой происходит обращение

Сложная привязка данных к элементам управления

Аналогично формируется привязка данных к объекту **ListBox**.

Значение поля, указанного в свойстве элемента управления **ValueMember**, соответствующее выбранному пользователем элементу, хранится в свойстве **SelectValue** элемента управления **ListBox** или **ComboBox**. Оно отличается от значения свойства **SelectItem**, которое хранит значение поля, указанного в свойстве **DisplayMember**.

Примеры привязки данных к списку и раскрывающемуся списку соответственно:

```
listBox1.DataSource=ds1.Tables["Автомобиль"];
listBox1.DisplayMember="Марка";
listBox1.ValueMember="Марка";

comboBox1.DataSource=ds1.Tables["Автомобиль"];
comboBox1.DisplayMember="Марка";
comboBox1.ValueMember="Марка";
```

Особенности привязки данных к элементу управления DataGrid

Особенность привязки данных к элементу управления **DataGrid** заключается в том, что разработчик имеет дело с более обширным множеством данных.

Элемент управления **DataGrid** позволяет установить значение свойства **DataSource** равным фактическому источнику данных (такому, как объект **DataTable**, **DataSet**, **DataView** и т.п.). Если изменить значение свойства **DataSource** на этапе выполнения, оно вступит в силу только после перезагрузки элемента управления **DataGrid**, осуществляемой с помощью метода **DataDrid.SetDataBinding().**

В качестве параметров метод **SetDataBinding()** принимает новые значения свойств **DataSource** и **DataMember**. Значение свойства **DataMember** – это именованная сущность, с которой будет связан элемент управления **DataGrid**.

§7 Пример создания Windows-приложения

Постановка задачи

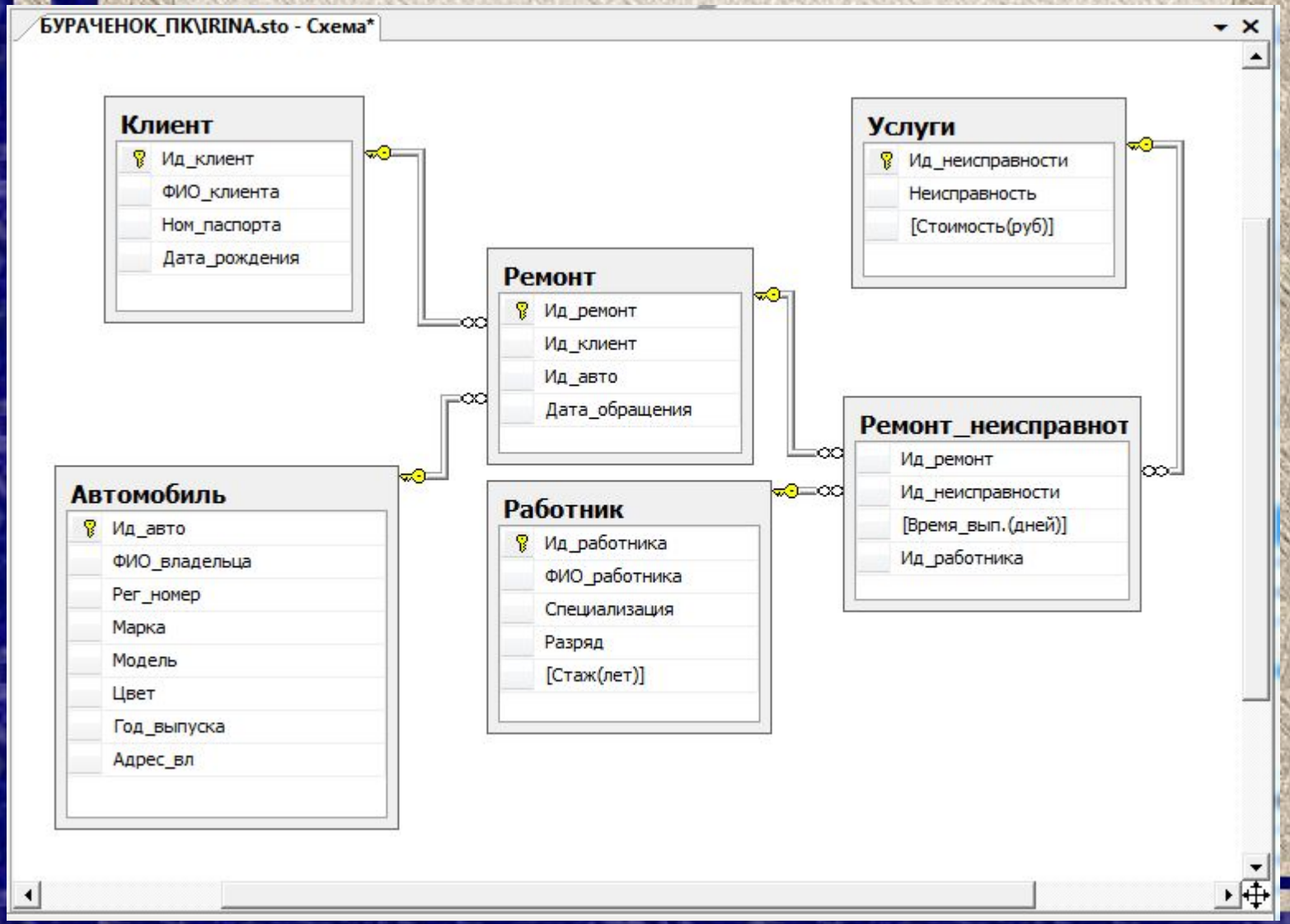
Пусть требуется создать программную систему, предназначенную для диспетчера станции техобслуживания. Такая система должна обеспечить хранение данных об:

- услугах, оказываемых станцией и их стоимости;
- клиентах станции, сдающих свои автомобили на ремонт;
- автомобилях клиентов;
- работниках станции.

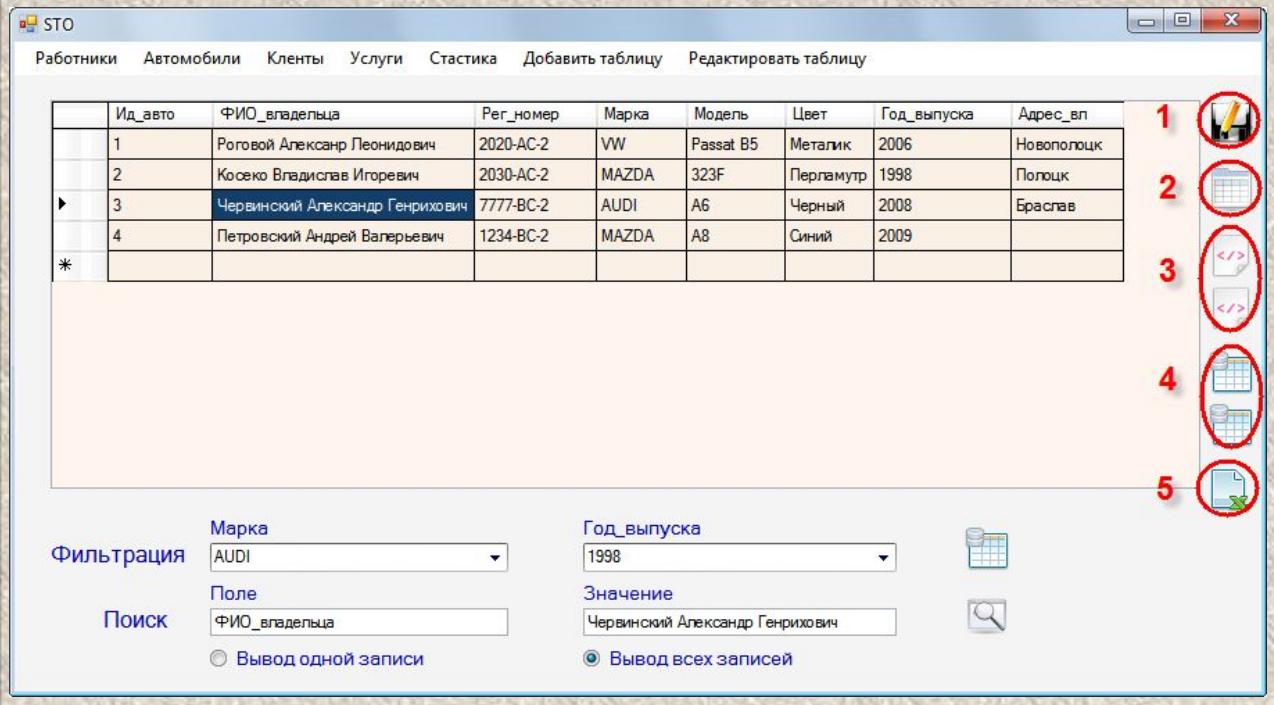
Также в программной системе должны быть реализованы следующие функции:

- поиск,
- фильтрация данных,
- добавление таблицы из приложения,
- экспорт данных в XML, XLS.

Схема базы данных STO



Внешний вид Windows-приложения может быть таким:



Программный код подключения к БД **STO**:

```
try
{
//Создаем объект поключения к БД
SqlConnection cn = new SqlConnection();
//Инициализируем свойство ConnectionString строкой подключения к БД
cn.ConnectionString = "data source=\"(local)\";initial
catalog=\"STO\"; IntegratedSecurity=true";
//Открываем созданное ранее подключение
cn.Open();
}
//Если подключение выполнить не удалось
//то обрабатываем соответствующее исключение
catch (Exception)
{
MessageBox.Show("SQL сервер не запущен!");
}
```

Программный код заполнения **DataSet** базы данных **STO**

```
DataSet Sto = new DataSet("Sto");//Для хранения всей БД «STO»
SqlDataAdapter AvtoAdap; //Адптер данных для таблицы «Автомобиль»
SqlDataAdapter KlientAdap; //Адптер данных для таблицы «Клиент»
SqlDataAdapter RabotnikAdap; //Адптер данных для таблицы «Работник»
SqlDataAdapter UsligiAdap; //Адптер данных для таблицы «Услуги»
SqlCommandBuilder AvtoBild; //Построитель команд для таб. «Автомобиль»
SqlCommandBuilder KlientBild; //Построитель команд для таб. «Клиент»
SqlCommandBuilder RabotnikBild; //Построитель ком. для таб. «Работник»
SqlCommandBuilder UslugiBild; // Построитель команд для таб. «Услуги»
//Функция заполения DataSet данными из таблиц БД «STO»
//Возвращаемые значения: 1 – в случае успешного заполения
//0 –в противном случае
```

Продолжение

```
private int ReadTable()
{
try
{
//Передаем  в адаптер запрос на выбоку всей таблицы «Автомобиль»
AvtoAdap = new SqlDataAdapter("SELECT * FROM Автомобиль", cn);
AvtoBild = newSqlCommandBuilder(AvtoAdap);
//Сохранение содержимого таблицы «Автомобиль» в DataSet
AvtoAdap.Fill(Sto, "Автомобиль");

//Передаем  в адаптер запрос на выбоку всей таблицы «Работник»
RabotnikAdap = newSqlDataAdapter("SELECT * FROM Работник", cn);
RabotnikBild= newSqlCommandBuilder(RabotnikAdap);
//Сохранение содержимого таблицы «Работник» в DataSet
RabotnikAdap.Fill(Sto, "Работник");

//Передаем  в адаптер запрос на выбоку всей таблицы «Клиент»
KlientAdap = newSqlDataAdapter("SELECT * FROM Клиент",cn);
KlientBild = newSqlCommandBuilder(KlientAdap);
//Сохранение содержимого таблицы «Клиент» в DataSet
KlientAdap.Fill(Sto, "Клиент");

//Передаем  в адаптер запрос на выбоку всей таблицы «Услуги»
UslugiAdap = newSqlDataAdapter("SELECT * FROM Услуги",cn);
UslugiBild = newSqlCommandBuilder(UsligiAdap);
//Сохранение содержимого таблицы «Услуги» в DataSet
UsligiAdap.Fill(Sto, "Услуги");
return 1;
}
catch (Exception)
{
MessageBox.Show("Ошибка заполнения!");
return 0;
}
}
```

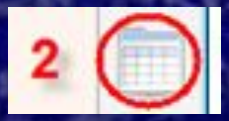
Программный код вывода соответствующей таблицы из **DataSet** базы данных **STO**

```
Private void работникиToolStripMenuItem_Click(object sender, EventArgs e)
{
            Table = 2;
            //В источник данных dataGridView1 передаем объект
            //DataTable «Работник» из DataSet «STO»
            dataGridView1.DataSource = Sto.Tables["Работник"];
}

Private void автомобилиToolStripMenuItem_Click(object sender, EventArgs e)
{
            Table = 1;
            //В источник данных dataGridView1 передаем объект
            //DataTable «Автомобиль» из DataSet «STO»
            dataGridView1.DataSource = Sto.Tables["Автомобиль"];
}


Private void клентыToolStripMenuItem_Click(object sender, EventArgs e)
{
            Table = 3;
            //В источник данных dataGridView1 передаем объект
            //DataTable «Клиент» из DataSet «STO»
            dataGridView1.DataSource = Sto.Tables["Клиент"];
}

Private void услугиToolStripMenuItem_Click(object sender, EventArgs e)
{
            Table = 4;
            //В источник данных dataGridView1 передаем объект
            //DataTable «Услуги» из DataSet «STO»
            dataGridView1.DataSource = Sto.Tables["Услуги"];
}
```

Программный код внесения измененных данных из **DataSet STO** на сервер БД (см. рисунок, пункт 1):


```
Private void UpdateDB()
{
  try
  {
     if (Table == 1)
                    //У адаптера данных вызываем метод сохранения
                    //данных таблицы «Автомобиль» на сервере БД
                    AvtoAdap.Update(Sto.Tables["Автомобиль"]);
     if (Table == 2)
                    //У адаптера данных вызываем метод сохранения
                    //данных таблицы «Работник» на сервере БД
                    RabotnikAdap.Update(Sto, "Работник");
     if (Table == 3)
                    //У адаптера данных вызываем метод сохранения
                    //данных таблицы «Клиент» на сервере БД
                    KlientAdap.Update(Sto, "Клиент");
     if (Table == 4)
                    //У адаптера данных вызываем метод сохранения
                    //данных таблицы «Услуги» на сервере БД
                    UsligiAdap.Update(Sto, "Услуги");
     MessageBox.Show("Изменения в базе данных выполнены!");
  }
  catch (Exception)
  {
     MessageBox.Show("Изменения в базе данных  не выполнены!");
  }
}
Private void pictureBox2_Click(object sender, EventArgs e)
{
            UpdateDB();
}
```

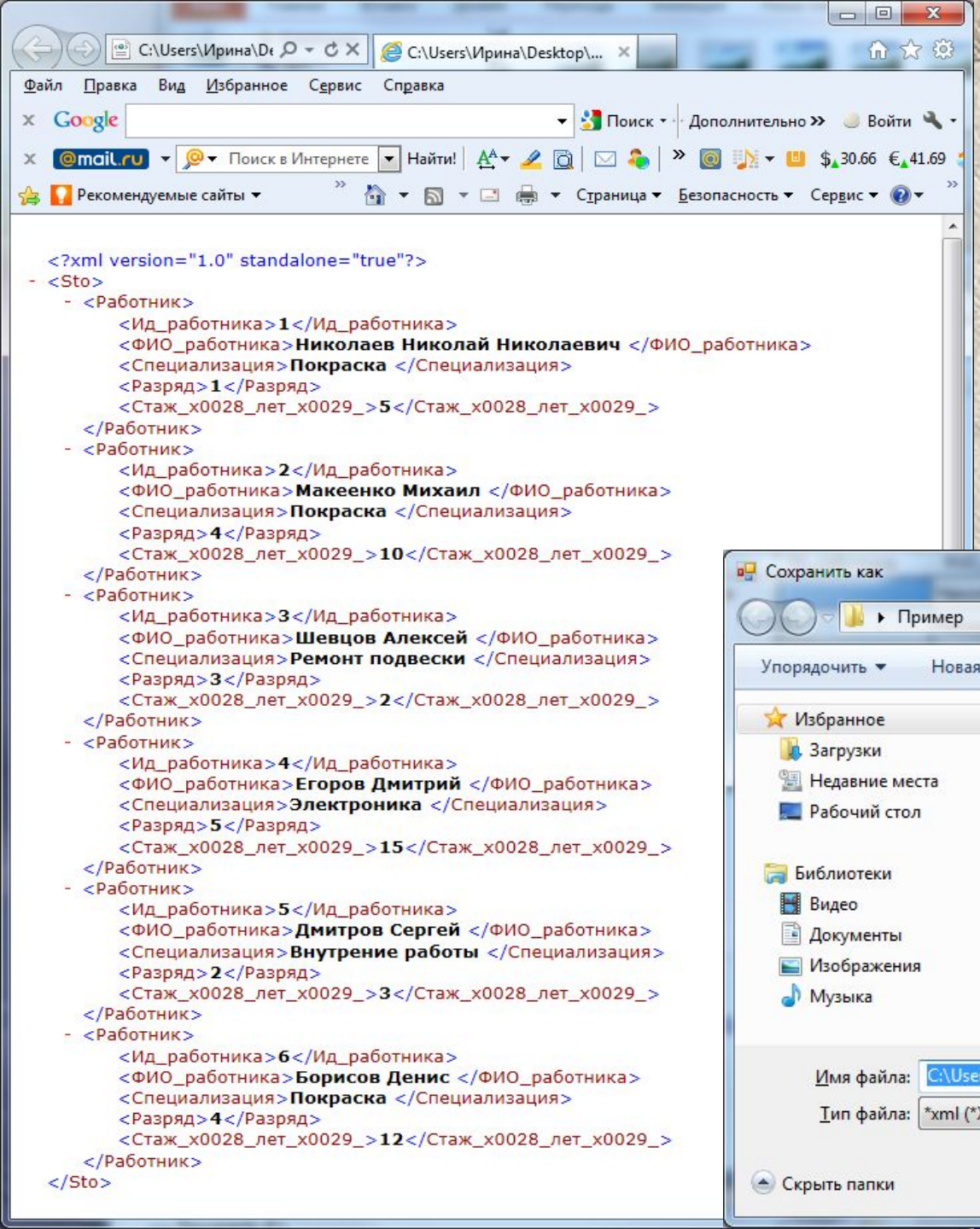
Программный код вывода содержимого всей таблице после фильтрации (см. рисунок, пункт 2):



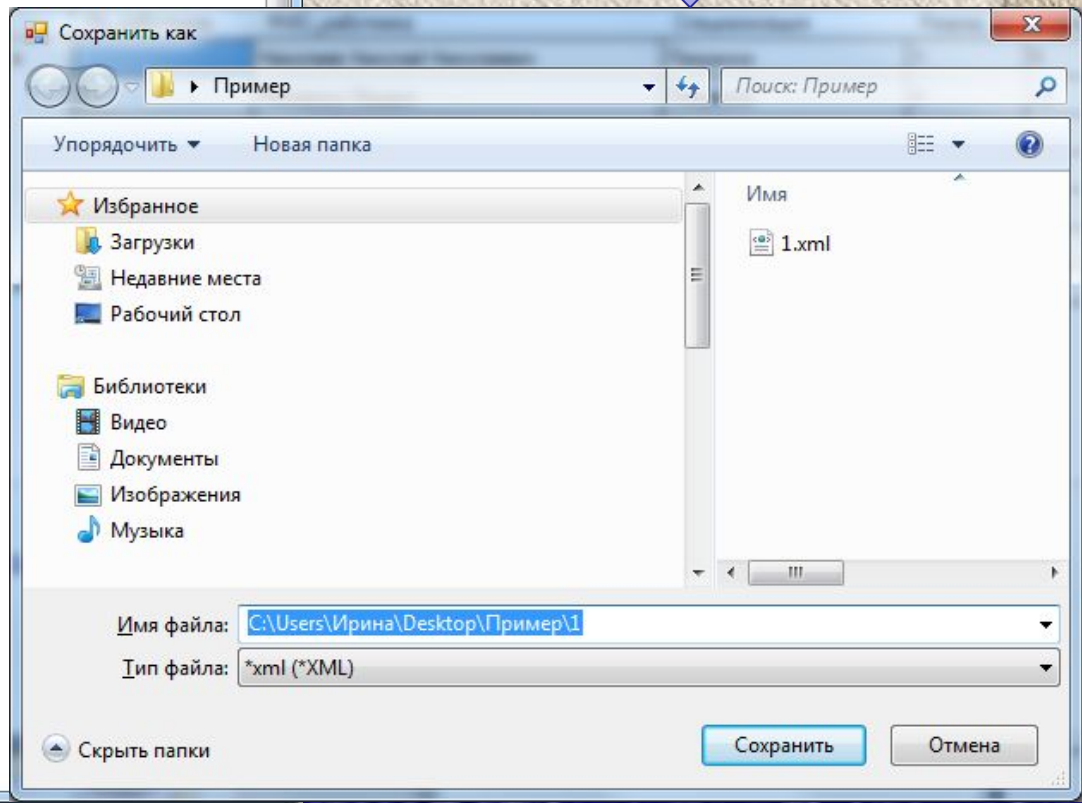

```
Private void pictureBox3_Click(object sender, EventArgs e)
{
if (Table == 1)
{
     //В источник данных dataGridView1 передаем объект
     //DataTable «Автомобиль» из DataSet «STO»
     dataGridView1.DataSource = Sto.Tables["Автомобиль"];
}
if (Table == 2)
{
     //В источник данных dataGridView1 передаем объект
     //DataTable «Работник» из DataSet «STO»
    dataGridView1.DataSource = Sto.Tables["Работник"];
}
}
```

Программный код экспорта содержимого таблицы в XML (см. рисунок, пункт 3):

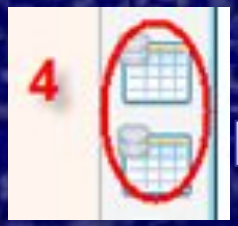

```
Private void pictureBox5_Click(object sender, EventArgs e)
{
saveFileDialog1.Filter = "*xml|*XML";
saveFileDialog1.ShowDialog();
while (saveFileDialog1.FileName == "") { }
if (Table == 1)
     Sto.Tables["Автомобиль"].WriteXml(saveFileDialog1.FileName
     + ".xml");
if (Table == 2)
     Sto.Tables["Работник"].WriteXml(saveFileDialog1.FileName
     + ".xml");
if (Table == 3)
     Sto.Tables["Клиент"].WriteXml(saveFileDialog1.FileName
     + ".xml");
if (Table == 4)
     Sto.Tables["Услуги"].WriteXml(saveFileDialog1.FileName
     + ".xml");
}

Private void pictureBox6_Click(object sender, EventArgs e)
{
            saveFileDialog1.Filter = "*xml|*XML";
            saveFileDialog1.ShowDialog();
            Sto.WriteXml(saveFileDialog1.FileName + ".xml");
}
```



Экспорт содержимого таблицы **Работник** в XML (см. рисунок, пункт 3):



Программный код экспорта содержимого таблицы в XSD (см. рисунок, пункт 4):

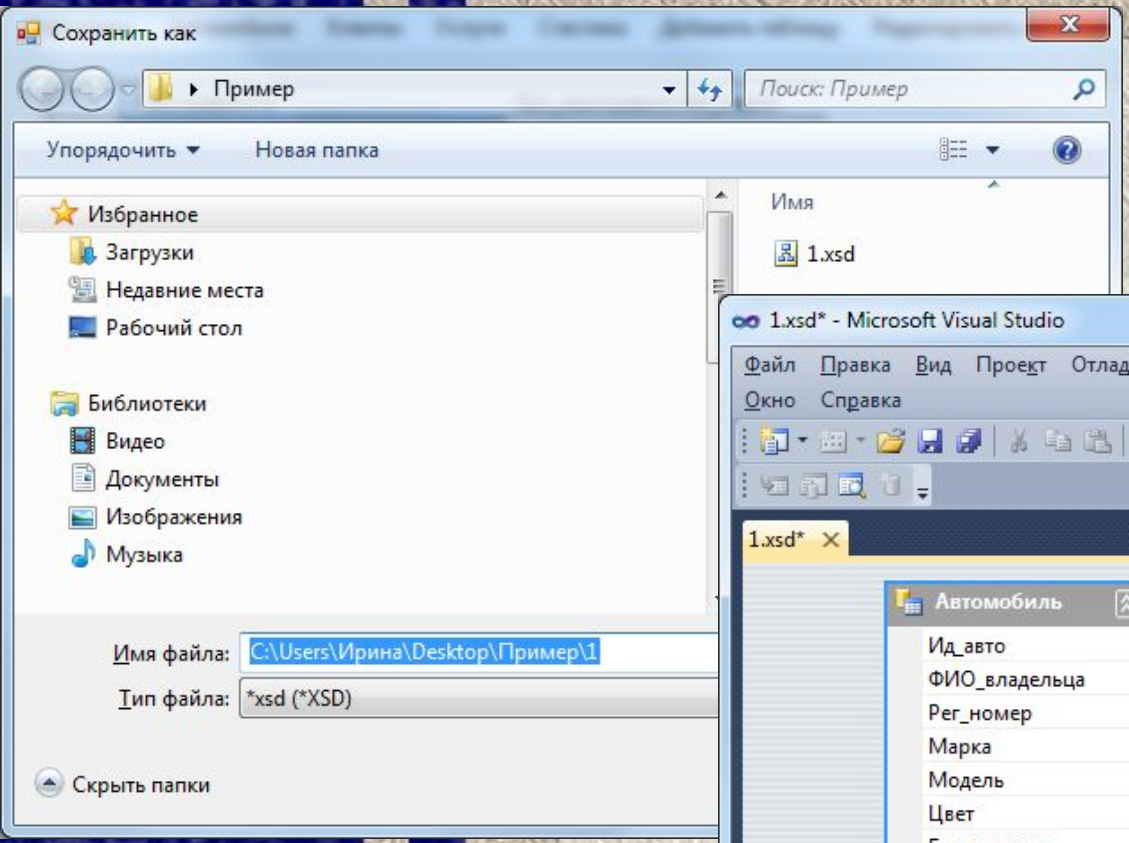


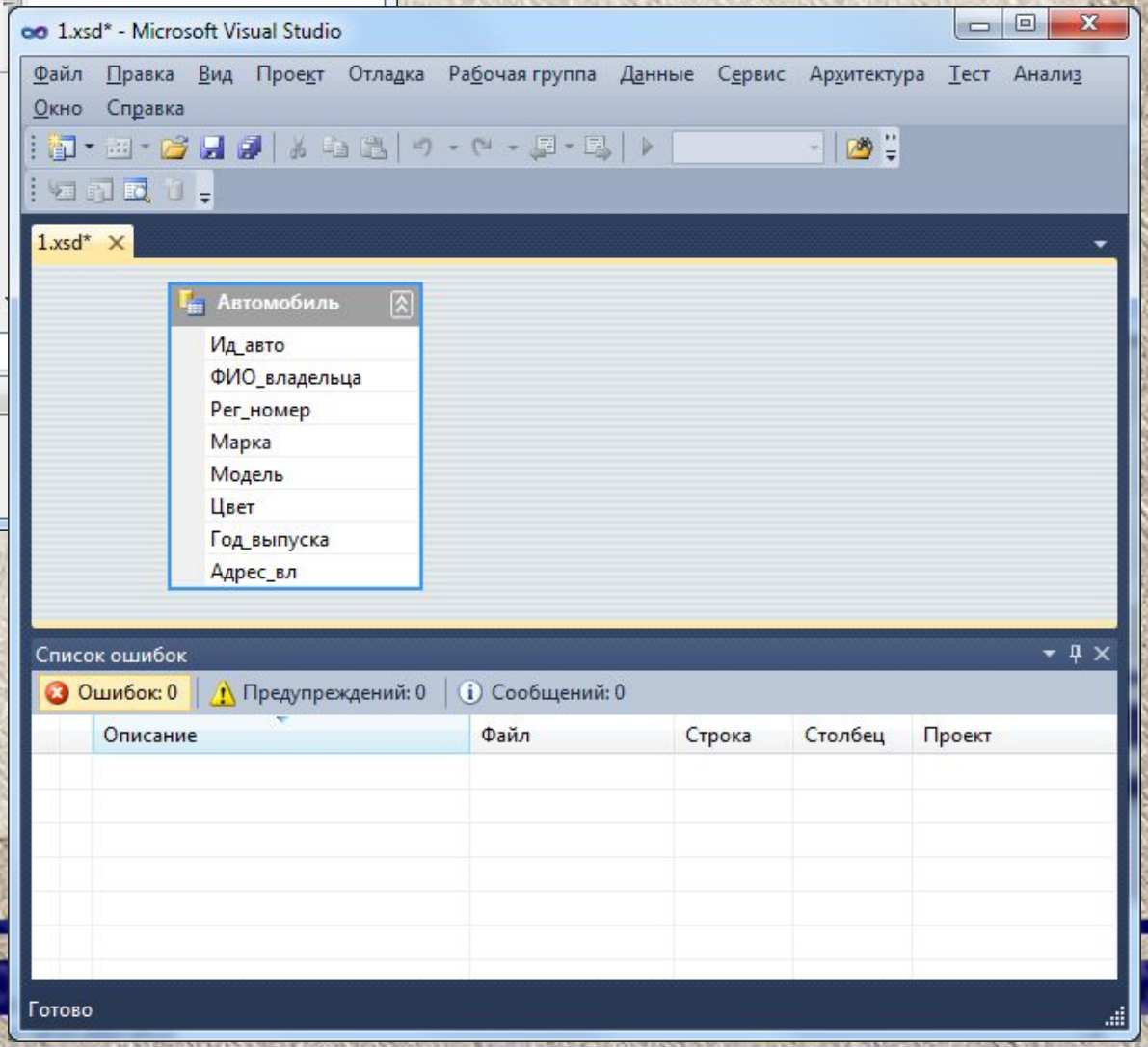

```
Private void pictureBox8_Click(object sender, EventArgs e)
{
            saveFileDialog1.Filter = "*xsd|*XSD";
            saveFileDialog1.ShowDialog();
            Sto.WriteXmlSchema(saveFileDialog1.FileName + ".XSD");
}
```

Программный код экспорта содержимого таблицы в XLS (см. рисунок, пункт 5):

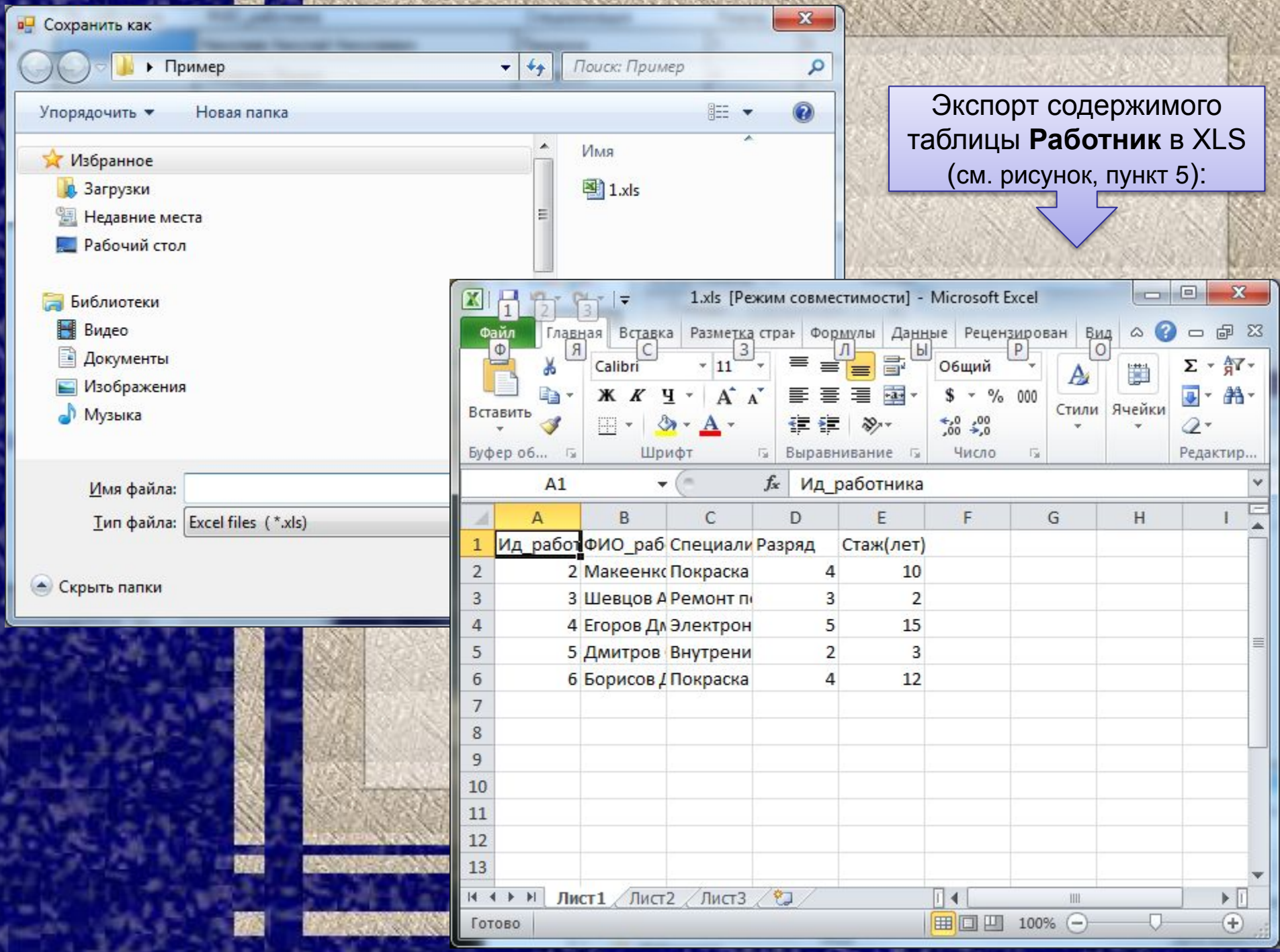

```
Private void pictureBox9_Click(object sender, EventArgs e)
{
            saveFileDialog1.Filter = "*xls|*XLS";
            saveFileDialog1.ShowDialog();
            Sto.WriteXmlSchema(saveFileDialog1.FileName + ".XLS");
}
```

Экспорт содержимого таблицы **Автомобиль** в XSD (см. рисунок, пункт 4):

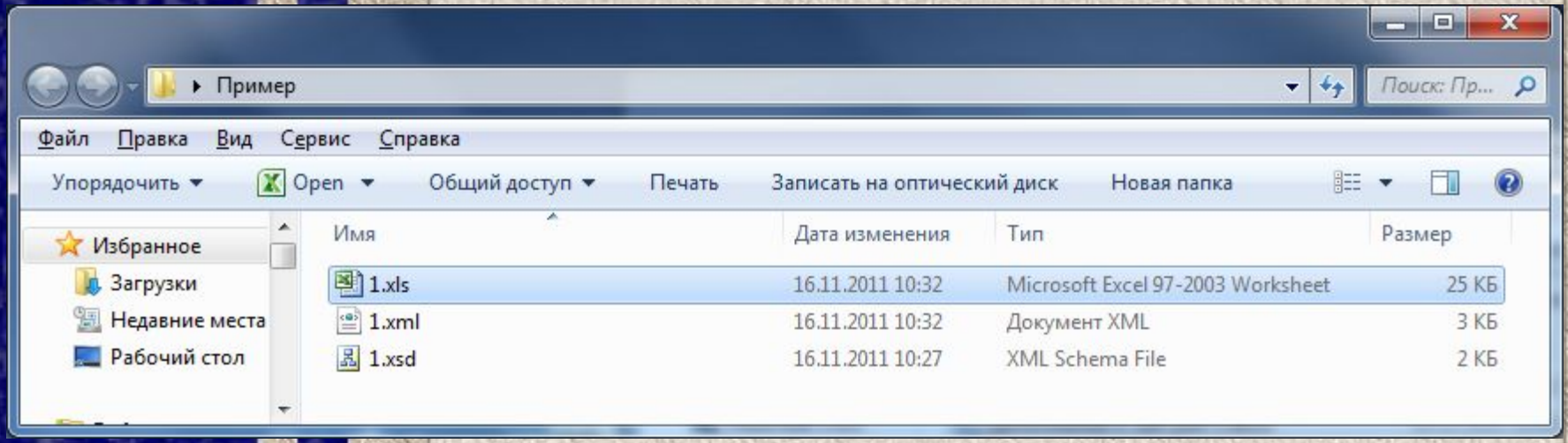




Экспорт содержимого таблицы **Работник** в XLS (см. рисунок, пункт 5):



Перечень полученных отчетов



Программный код запуска хранимой процедуры “MarkaNeisp” с параметром “Mar”, которая выводит неисправности указанной марки автомобиля:

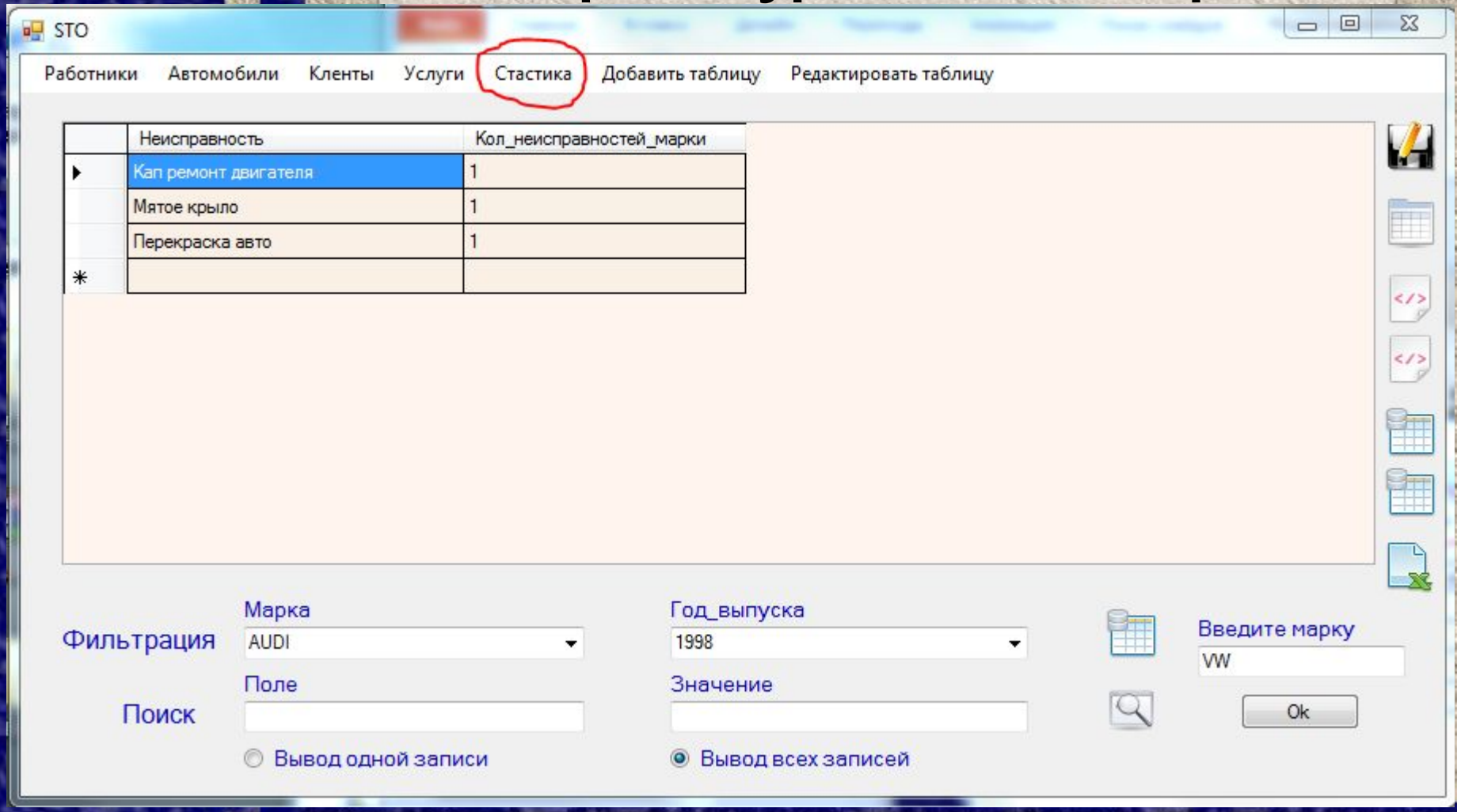
```
Private void button1_Click_1(object sender, EventArgs e)
{
//Создаем объект хранения данных полученных в результате выполнения
//запроса
DataTable FilTable = new DataTable();

//Вызываем хранимую процедуру из настроеннного ранее подключения cn
SqlCommand cmd = new SqlCommand("MarkaNeisp", cn);

//Обьект-комнда имеет тип хранимой процедуры
cmd.CommandType = CommandType.StoredProcedure;

//Указываем параметор ”Mar” типа char
cmd.Parameters.Add(new SqlParameter("@Mar", SqlDbType.Char));

//Передаем значение параметра хранимой процедуры
cmd.Parameters["@Mar"].Value = textBox3.Text;

//Обьект-команду передаем в адаптер данных
SqlDataAdapter FilAdap = newSqlDataAdapter(cmd);
/
/Сохраняем результат хранимой процедуры в обьект DataTable
FilAdap.Fill(FilTable);

//Выводим результат хранимой процедуры  в dataGridView1
dataGridView1.DataSource = FilTable;
}
```

Текст хранимой процедуры MarkaNeisp

SQLQuery3.sql - БУРАЧЕНОК_... (59))*

```
GO
CREATE PROCEDURE [dbo].[MarkaNeisp] @Mar CHAR(50)
AS
BEGIN
SELECT      TOP 100 PERCENT dbo.Услуги.Неисправность, COUNT(dbo.Услуги.Неисправность) AS [Кол_неисправностей_марки]
FROM          dbo.Автомобиль INNER JOIN
                      dbo.Ремонт ON dbo.Автомобиль.Ид_авто = dbo.Ремонт.Ид_авто
                      INNER JOIN
                      dbo.Ремонт_неисправноть ON dbo.Ремонт.Ид_ремонт = dbo.Ремонт_неисправноть.Ид_ремонт
                      INNER JOIN
                      dbo.Услуги ON dbo.Ремонт_неисправноть.Ид_неисправности = dbo.Услуги.Ид_неисправности
WHERE      (dbo.Автомобиль.Марка = @Mar)
GROUP BY dbo.Услуги.Неисправность
ORDER BY COUNT(dbo.Услуги.Неисправность) DESC
END
```

Реализация хранимой процедуры MarkaNeisp



Примеры фильтров