**Лабораторная работа № 10: Работа с базами данных — XML и OLE**

**Содержание**

1. [**Вводная часть**](#B01)
2. [**Создание приложения Windows Forms**](#B02)
3. [**Модификация приложения Windows Forms**](#B03)**: ODBC**
4. [**Модификация приложения Windows Forms: OLE**](#B04)
5. [**Модификация приложения Windows Forms: XML**](#B05)
6. [**Завершающая часть**](#B06)
7. [**О приложении к Лабораторной работе № 10**](#B07)

**1. Вводная часть**

В этой работе будет рассмотрена работа с базами данных. А именно с текстовыми файлами **\*.xml** и языком расширяемой разметки (от англ. **eXtensible Markup Language** – расширяемый язык разметки) **XML**, а также **ADO.NET** (**ActiveX Data Objects .NET**) и один из **провайдеров** данных ADO.NET: в частности **OLE** (**Object Linking and Embedding**) и **ODBC** (**Open Database Connectivity**).

По данной тематике (работе с базами данных в C#) на данный момент существует достаточное количество материалов, как в печатном издании, так и в сети Интернет (различных статей и курсов). В данной работе будут рассмотрены лишь основные обобщённые моменты работы с базами данных (через ODBC, OLE и XML).

Что такое ADO.NET?

*ActiveX Data Objects .NET* является набором классов, реализующих программные интерфейсы для облегчения подключения к базам данных из приложения независимо от особенностей реализации конкретной системы управления базами данных и от структуры самой базы данных, а также независимо от места расположения этой самой базы — в частности, в распределенной среде (клиент-серверное приложение) на стороне сервера.

ADO.NET широко используется совместно с технологией web-программирования с использованием объектов **ASP.NET** для доступа к расположенным на сервере базам данных со стороны клиента.

В ADO.NET используется доступ к отсоединенным данным. При этом соединение устанавливается лишь на то время, которое необходимо для проведения определенной операции над базой данных.

Поставщик данных для приложения (**Provider**) – объект, предназначенный для обеспечения взаимодействия приложения с хранилищем информации (базами данных).

Естественно, приложению нет никакого дела до того, где хранится и как извлекается потребляемая приложением информация. Для приложения источником данных является тот, кто передает данные приложению. И как сам этот источник эту информацию добывает – никого не касается.

Источник данных (**Data Provider**) – это набор взаимосвязанных компонентов, обеспечивающих доступ к данным. Функциональность и само существование провайдера обеспечивается набором классов, специально для этой цели разработанных.

ADO.NET поддерживает следующие типа источников данных:

|  |  |  |
| --- | --- | --- |
| **Имя провайдера** | **API-префикс** | **Описание источника данных** |
| ODBC Data Provider | Odbc | Источники данных с ODBC-интерфейсом. Устаревший провайдер. |
| OleDb Data Provider | OleDb | Источники данных с OleDb-интерфейсом, для Access или Excel. |
| Oracle Data Provider | Oracle | Для баз данных Oracle. |
| SQL Data Provider | Sql | Для работы с Microsoft SQL Server. |
| Borland Data Provider | Bdp | Общий доступ к множеству баз данных, таких как Interbase, SQL Server, IBM DB2, и Oracle. |

Нас пока интересует только *ODBC* и собственно *OLE*.

Что такое ODBC?

ODBC —это программный интерфейс (**API**) доступа к базам данных, разработанный фирмой Microsoft, в сотрудничестве с Simba Technologies на основе спецификаций Call Level Interface (CLI), который разрабатывался организациями SQL Access Group, X/Open и Microsoft. Впоследствии CLI был стандартизован ISO ISO/IEC 9075-3:2003. Стандарт CLI призван унифицировать программное взаимодействие с СУБД, сделать его независимым от поставщика СУБД и программно-аппаратной платформы.

В начале 1990 г. существовало несколько поставщиков баз данных, каждый из которых имел собственный интерфейс. Если приложению было необходимо общаться с несколькими источниками данных, для взаимодействия с каждой из баз данных было необходимо написать свой код. Для решения возникшей проблемы Microsoft и ряд других компаний создали стандартный интерфейс для получения и отправки источникам данных различных типов. Этот интерфейс был назван *Open Database Connectivity*, или открытый механизм взаимодействия с базами данных.

C помощью ODBC прикладные программисты могли разрабатывать приложения для использования одного интерфейса доступа к данным, не беспокоясь о тонкостях взаимодействия с несколькими источниками.

Это достигается благодаря тому, что поставщики различных баз данных создают драйверы, реализующие конкретное наполнение стандартных функций из ODBC API с учётом особенностей их продукта. Приложения используют эти функции, реализованные в соответствующем конкретному источнику данных драйвере, для унифицированного доступа к различным источникам данных.

**MFC** (**Microsoft Foundation Classes**, библиотека на языке С++, Microsoft) усовершенствовала ODBC для разработчиков приложений. Истинный интерфейс ODBC является обычным процедурным API. Вместо создания простой оболочки процедурного API разработчики MFC создали набор абстрактных классов, представляющих логические сущности в базе данных.

Что такое OLE?

OLE (произносится как oh-lay [олэй]) — технология связывания и внедрения объектов в другие документы и объекты, разработанная корпорацией Майкрософт.

В 1996 году Microsoft переименовала технологию в **ActiveX**.

OLE позволяет передавать часть работы от одной программы редактирования к другой и возвращать результаты назад. Например, установленная на персональном компьютере издательская система может послать некий текст на обработку в текстовый редактор, либо некоторое изображение в редактор изображений с помощью OLE-технологии.

Основное преимущество использования OLE (кроме уменьшения размера файла) в том, что она позволяет создать главный файл, картотеку функций, к которой обращается программа. Этот файл может оперировать данными из исходной программы, которые после обработки возвращаются в исходный документ.

OLE используется при обработке составных документов (**compound documents**), может быть использована при передаче данных между различными несвязанными между собой системами посредством интерфейса переноса (**drag-and-drop**), а также при выполнении операций с буфером обмена. Идея внедрения широко используется при работе с мультимедийным содержанием на веб-страницах (пример — Веб-ТВ), где используется передача изображения, звука, видео, анимации в страницах HTML (язык гипертекстовой разметки) либо в других файлах, также использующих текстовую разметку (например, XML и **SGML**). Однако, технология OLE использует архитектуру «толстого клиента», то есть сетевой ПК с избыточными вычислительными ресурсами. Это означает, что тип файла либо программа, которую пытаются внедрить, должна присутствовать на машине клиента. Например, если OLE оперирует таблицами Microsoft Excel, то программа Excel должна быть инсталлирована на машине пользователя.

Что такое XML?

XML (произносится [экс-эм-э́л]) — рекомендованный **Консорциумом Всемирной паутины** (**W3C**) язык разметки, фактически представляющий собой свод общих синтаксических правил. XML — текстовый формат, предназначенный для хранения структурированных данных (взамен существующих файлов баз данных), для обмена информацией между программами, а также для создания на его основе более специализированных языков разметки (например, **XHTML**). XML является упрощённым подмножеством языка SGML. Подробнее об XML будет рассказано ниже.

Что будет представлять собой приложение, разрабатываемое в данной работе?

Приложение будет представлять собой окно **Windows Forms**, поделённое на группы элементов, в каждой группе будут располагаться элементы управления для работы с той или иной задачей и базой данных.

**2. Создание приложения Windows Forms**

Запускаем Visual Studio 2010, откроется **Начальная страница**:

Для начала, надо создать проект, для этого выполним последовательно: **Файл** -> **Создать** -> **Проект**… (также можно просто нажать сочетание клавиш **Ctrl+Shift+N** или пункт «**Создать проект**…» на *Начальной странице*):

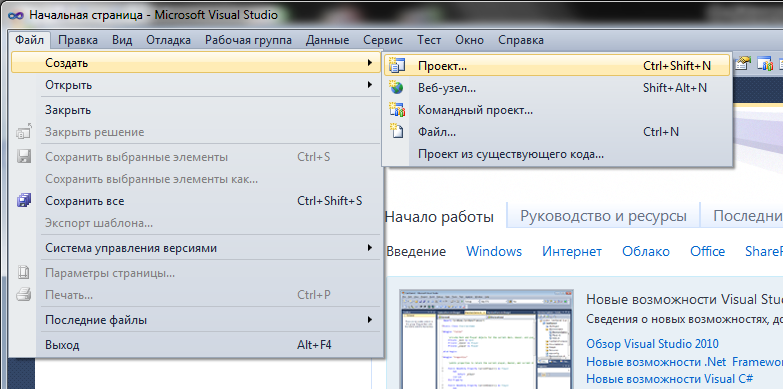


Рис. 2. 1. Создание нового проекта

Выберем слева в пункте **Установленные шаблоны** язык **Visual C#**, далее найдём в списке **Приложение Windows Forms**. Также здесь можно выбрать какой использовать «фреймворк» (набора компонентов для написания программ). В нашем случае выберем **.NET Framework 4**.

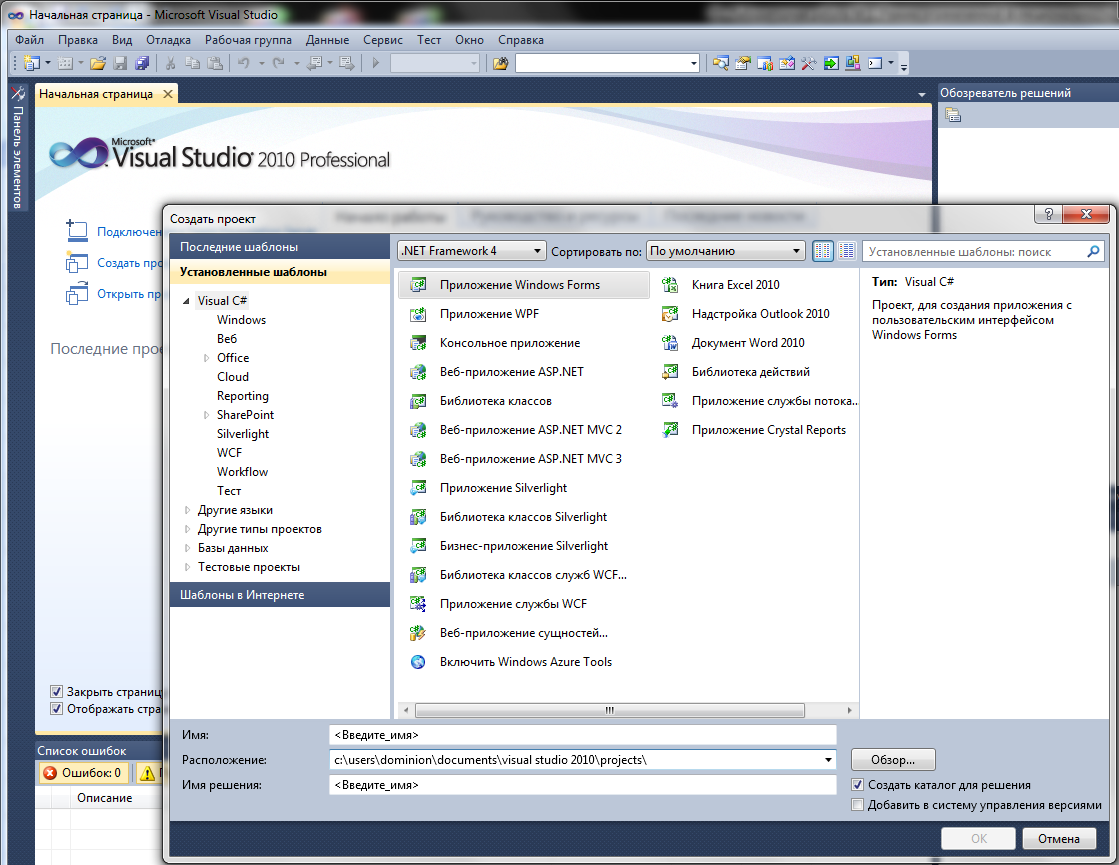


Рис. 2. 2. Окно создания нового проекта

В поле **Имя** вводим **LWP10DB** **—** это название программы (выбрано по названию лабораторного практикума, номеру и названию работы). В поле **Расположение** указана конечная директория, где будет находиться весь проект. Выберем расположение удобное для быстрого поиска. В поле **Имя решения** вводится либо название программы «по умолчанию» из поля *Имя* автоматически, либо можно ввести своё собственное. Под этим именем будет создана конечная папка проекта (если *Имя* и *Имя решения* разные).

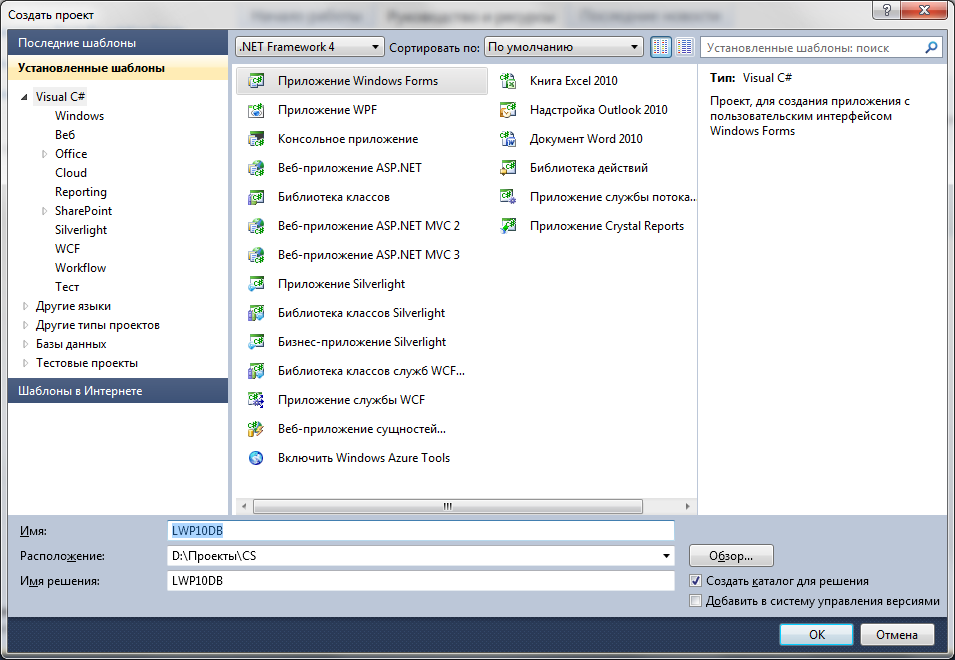


Рис. 2. 3. Вводим данные нового проекта приложения *Windows Forms*

После нажатия клавиши **ОК** мы увидим сформированный проект и исходный код приложения *Windows Forms* (не пустого изначально).

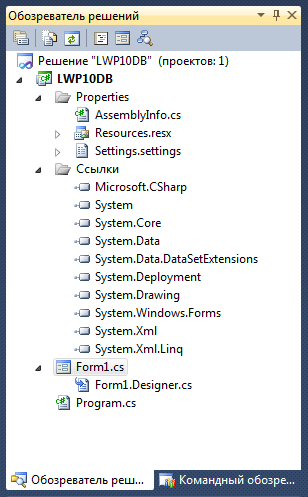


Рис. 2. 4. **Обозреватель решений**: состав проекта приложения *Windows Forms* сформированного средой разработки

Теперь, можно откомпилировать созданную программу, нажав клавишу **F5** (**Отладка** -> **Начать отладку** или нажав на иконку. Тем самым мы запускаем приложение в режиме отладки (и производим компиляцию debug-версии программы) (**Debug** выбрано изначально).

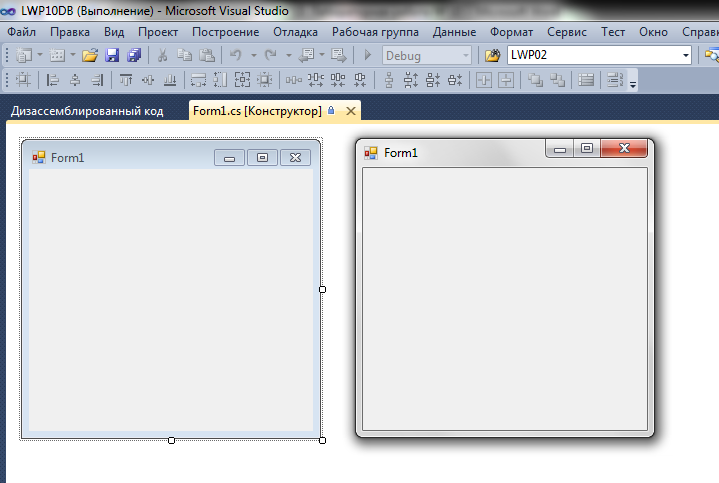


Рис. 2. 5. Запуск приложения *Windows Forms* по конфигурации *Debug*

**3. Модификация приложения Windows Forms: ODBC**

Для начала изменим размер нашей единственной формы. Для этого можно потянуть за уголок в нужном направлении на странице визуального представления формы**1**. Но также размер можно менять на панели свойств этой формы. Для этого нужно поменять значение размера в пикселях (высоту и ширину) в поле **Size**.

**ПРИМЕЧАНИЕ № 1**: Для перехода на визуальное представление формы, необходимо двойным нажатием в обозревателе решений нажать на значок формы () или выбрать вкладку на панели вкладок с именем **<имя формы>.cs [Конструктор]**.

Задаём следующие параметры формы на панели **Свойства**:

|  |  |
| --- | --- |
| **(Name)** | изменим с **Form1.cs2** на **LWP10Main** |

^ Поменяем внутреннее имя формы.

|  |  |
| --- | --- |
| **Text** | изменим с **Form1** на **Работа с базами данных (C#)** |

^ Поменяем заголовок формы (то что отображается в шапке приложения слева).

|  |  |
| --- | --- |
| **Icon** | изменим изображение (иконку) приложения |

^ Необходим файл значка **\*.ico**.

|  |  |
| --- | --- |
| **Size** | изменим со значений **300; 300** на **800; 600** |

^ Поменяем размер формы.

**ПРИМЕЧАНИЕ № 2**: Для того, чтобы поменять имя файла нашей формы, необходимо выполнить следующее: выделить в обозревателе решений значок формы () и нажать правую кнопку мыши, затем выбрать **Переименовать**. Ввести необходимое новое имя **СОХРАНЯЯ** расширение **\*.cs**. После смены имени, автоматически поменяются имена проассоциированных непосредственно с формой файлов:



Получим нечто подобное:

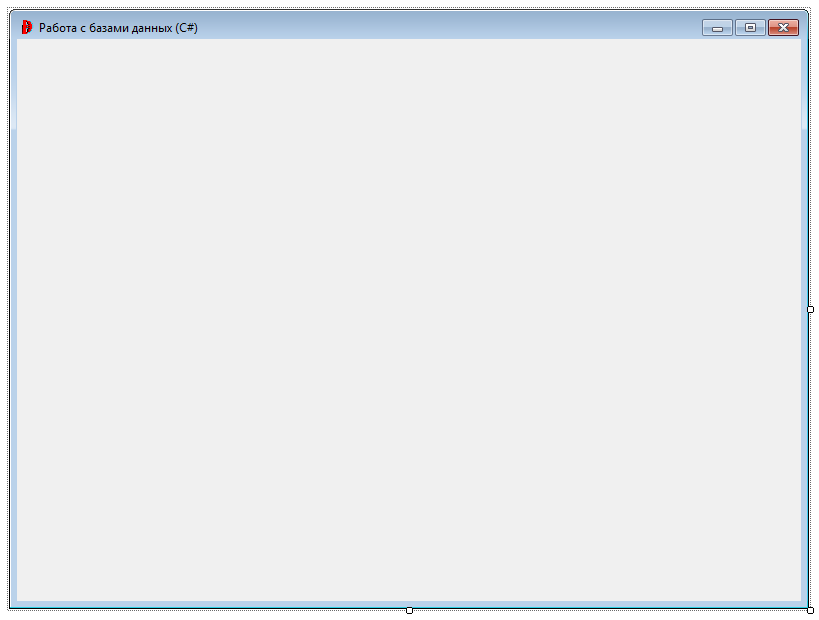


Рис. 3. 1. Модифицированная форма приложения

Добавим на нашу форму **ToolTip** ().

Параметры добавленного элемента всплывающей подсказки таковы:

|  |  |
| --- | --- |
| **(Name)**: | Hint |

Теперь приступим к формированию первой группы элементов и её функциональности. Для начала расставим элементы как показано на рисунке ниже:

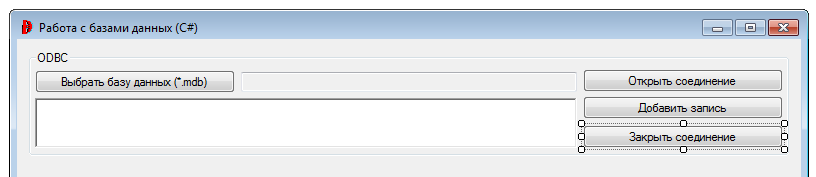


Рис. 3. 2. Расстановка элементов первой группы (**ODBC**)

Здесь у нас четыре кнопки **Button**, один **RichTextBox** (слева внизу), и простой погашенный **TextBox**.

По порядку распишем **Свойства** каждого элемента:

*Button:*

|  |  |
| --- | --- |
| **(Name)**: | B\_ODBC\_Search |
| **Text**: | Выбрать базу данных (\*.mdb) |
| **Size**: | 200; 23 |

*Button:*

|  |  |
| --- | --- |
| **(Name)**: | B\_ODBC\_Connect |
| **Text**: | Открыть соединение |
| **Size**: | 200; 23 |

*Button:*

|  |  |
| --- | --- |
| **(Name)**: | B\_ODBC\_Add |
| **Text**: | Добавить запись |
| **Size**: | 200; 23 |

*Button:*

|  |  |
| --- | --- |
| **(Name)**: | B\_ODBC\_Disconnect |
| **Text**: | Закрыть соединение |
| **Size**: | 200; 23 |

*TextBox:*

|  |  |
| --- | --- |
| **(Name)**: | TB\_ODBC\_Path |
| **ReadOnly**: | True |

*RicTextBox:*

|  |  |
| --- | --- |
| **(Name)**: | RTB\_ODBC |
| **ReadOnly**: | True |

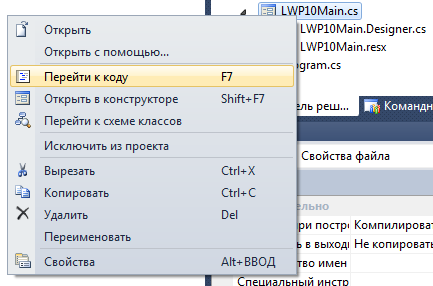
*GroupBox:*

|  |  |
| --- | --- |
| **(Name)**: | GB\_ODBC |
| **Text**: | ODBC |

*OpenFileDialog:*

|  |  |
| --- | --- |
| **(Name)**: | OFD\_ODBC |
| **FileName**: | LWP10-DB-ODBC |
| **InitialDirectory**: | D:\ |
| **Filter** | База данных \*.mdb|\*.mdb |

Теперь отправляемся в код формы (правая кнопка мыши на значке формы, далее Перейти к коду или нажмём на клавишу **F7**):



В самое начало кода добавим:

using System.Data.Odbc; // ODBC

Найдём:

public partial class LWP10Main : Form

{

Добавим после:

Double Counter = 4;

OdbcConnection ConnectionOBDC;

Изменим следующую функцию **LWP10Main()**:

public LWP10Main()

{

InitializeComponent();

B\_ODBC\_Add.Enabled = false;

B\_ODBC\_Connect.Enabled = false;

B\_ODBC\_Disconnect.Enabled = false;

TB\_ODBC\_Path.Text = "D:\\LWP10-DB-ODBC.mdb";

}

Событие **Click** кнопки **B\_ODBC\_Search** («Выбрать базу данных»):

private void B\_ODBC\_Search\_Click(object sender, EventArgs e)

{

if (OFD\_ODBC.ShowDialog() == DialogResult.OK)

{

B\_ODBC\_Add.Enabled = true;

B\_ODBC\_Connect.Enabled = true;

B\_ODBC\_Disconnect.Enabled = true;

Directory.CreateDirectory(Path.GetDirectoryName(OFD\_ODBC.FileName) + @"\Копии"); // Создаём директорию под изменённые БД

File.Copy(OFD\_ODBC.FileName, Path.GetDirectoryName(OFD\_ODBC.FileName) + @"\Копии\" + OFD\_ODBC.SafeFileName, true); // Копируем туда выбранную БД (перезаписываем, в случае обнаружения похожего файла)

if (Path.GetDirectoryName(OFD\_ODBC.FileName) == Directory.GetDirectoryRoot(OFD\_ODBC.FileName)) // Проверяем путь, если находимся в корневой директории диска, режем один слеш

TB\_ODBC\_Path.Text = Path.GetDirectoryName(OFD\_ODBC.FileName) + @"Копии\" + OFD\_ODBC.SafeFileName;

else

TB\_ODBC\_Path.Text = Path.GetDirectoryName(OFD\_ODBC.FileName) + @"\Копии\" + OFD\_ODBC.SafeFileName;

}

}

Небольшое замечание по коду выше. После выбора БД в окне диалога, приложение будет создавать в директории с БД новую папку «Копии» и копировать туда выбранную в диалоге БД. Все операции с БД (из нашего приложения) будут происходить с копией, а не с оригиналом. Копия БД будет затираться всякий раз при открытии диалога выбора и выбора там БД.

Событие *Click* кнопки **B\_ODBC\_Connect** («Открыть соединение»):

private void B\_ODBC\_Connect\_Click(object sender, EventArgs e)

{

String ConnetionStringODBC = null;

ConnetionStringODBC = "Driver={Microsoft Access Driver (\*.mdb)}; DBQ=" + TB\_ODBC\_Path.Text + ";"; // Выбираем источник данных ("провайдера") и указываем путь к нему через TextBox

ConnectionOBDC = new OdbcConnection(ConnetionStringODBC); // Инициализируем объект соединения с новыми параметрами

try

{

ConnectionOBDC.Open(); // Открываем соединение

MessageBox.Show("Соединение с базой данных " + TB\_ODBC\_Path.Text + " успешно открыто!", "Работа с базами данных (C#) :: ODBC");

}

catch (Exception ex)

{

MessageBox.Show("Невозможно открыть соединение с базой данных " + TB\_ODBC\_Path.Text + " (" + ex.Message + ")!", "Работа с базами данных (C#) :: ODBC");

}

}

Событие *Click* кнопки **B\_ODBC\_Add** («Добавить запись»):

private void B\_ODBC\_Add\_Click(object sender, EventArgs e)

{

String SQL\_ODBC = "INSERT INTO \"Главная таблица\" VALUES( '" + Counter++ + "', 'Число', '999', 'ABC' );"; // Запрос на на добавление записей в нашу таблицу, ключевое поле будет числовым, начинается с 4 и далее растёт инкрементом

OdbcCommand Command = new OdbcCommand(SQL\_ODBC, ConnectionOBDC); // Формируем команду

try

{

Command.ExecuteNonQuery(); // Выполняем команду

RTB\_ODBC.Clear(); // Очищаем RichTextBox

RTB\_ODBC.AppendText(Command.CommandText); // Вставляем результат выполнения команды с нашей базой

}

catch (Exception ex)

{

RTB\_ODBC.Clear();

RTB\_ODBC.AppendText(ex.Message);

}

}

Событие *Click* кнопки **B\_ODBC\_Disconnect** («Закрыть соединение»):

private void B\_ODBC\_Disconnect\_Click(object sender, EventArgs e)

{

String SQL\_ODBC = "DELETE FROM \"Главная таблица\" WHERE \"Главная таблица\".\"Первое поле\" = 'Число';"; // Запрос на удаление всего добавленного (чтобы не делать это вручную потом)

OdbcCommand Command = new OdbcCommand(SQL\_ODBC, ConnectionOBDC); // Формируем команду

try

{

Command.ExecuteNonQuery(); // Выполняем команду

RTB\_ODBC.Clear(); // Очищаем RichTextBox

RTB\_ODBC.AppendText(Command.CommandText); // Вставляем результат выполнения команды с нашей базой

ConnectionOBDC.Close(); // Закрываем соединение

MessageBox.Show("Соединение с базой данных " + TB\_ODBC\_Path.Text + " успешно закрыто!", "Работа с базами данных (C#) :: ODBC");

}

catch (Exception ex)

{

RTB\_ODBC.Clear();

RTB\_ODBC.AppendText(ex.Message);

MessageBox.Show("Невозможно закрыть соединение с базой данных " + TB\_ODBC\_Path.Text + " (" + ex.Message + ")!", "Работа с базами данных (C#) :: ODBC");

}

}

Последнее что нам нужно, это база данных в формате **Microsoft Access 2000**. Сделаем её, например в **Microsoft Office Access 2010**. База будет содержать одну таблицу (**Главная таблица**) и четыре столбца: **Ключевое поле** (являющее ключевым, числовое), : **Первое поле**, **Второе поле** и **Третье поле** (все текстовые).

Заполним первые три записи (Ключевое поле: 1, 2 и 3):

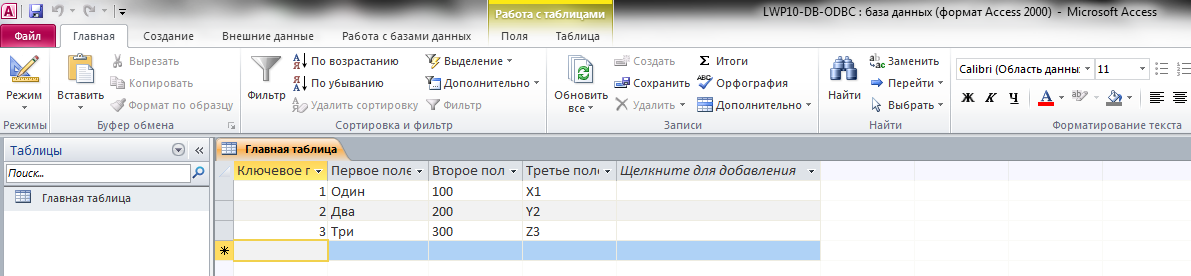


Рис. 3. 3. База данных **LWP10-DB-ODBC.mdb**

Для сохранения в формат Access 2000 выполним: **Файл** -> **Сохранить и опубликовать**, далее выберем формат Базы данных Access 2000:

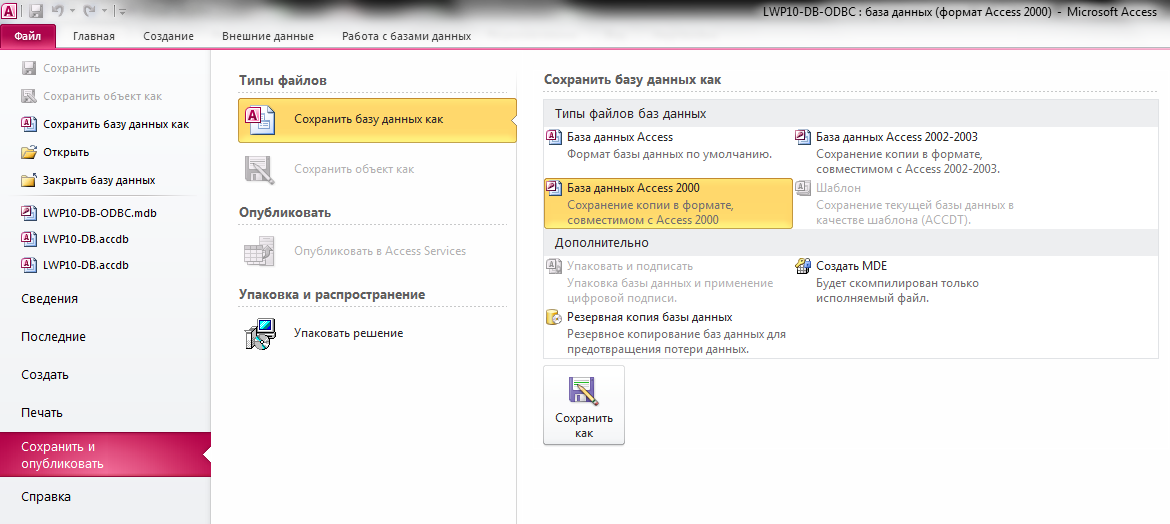
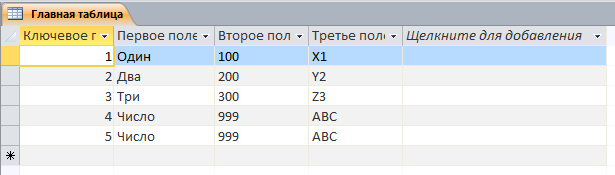


Рис. 3. 4. Сохранение в формате Access 2000

Компилируем приложение (**Debug**) и запускаем. Выбираем нашу базу кнопкой в левом верхнем углу приложения, затем жмём на **Открыть соединение**, далее несколько раз на **Добавить запись** (при этом наблюдая за тем что пишет *RichTextBox*). Если после нескольких добавлений открыть базу данных (не нажимая на **Закрыть соединение** или не закрывая само приложение), то можно увидеть новые записи:



Удаляем все новые записи кнопкой *Закрыть соединение* и тем самым также закрываем соединение с базой.

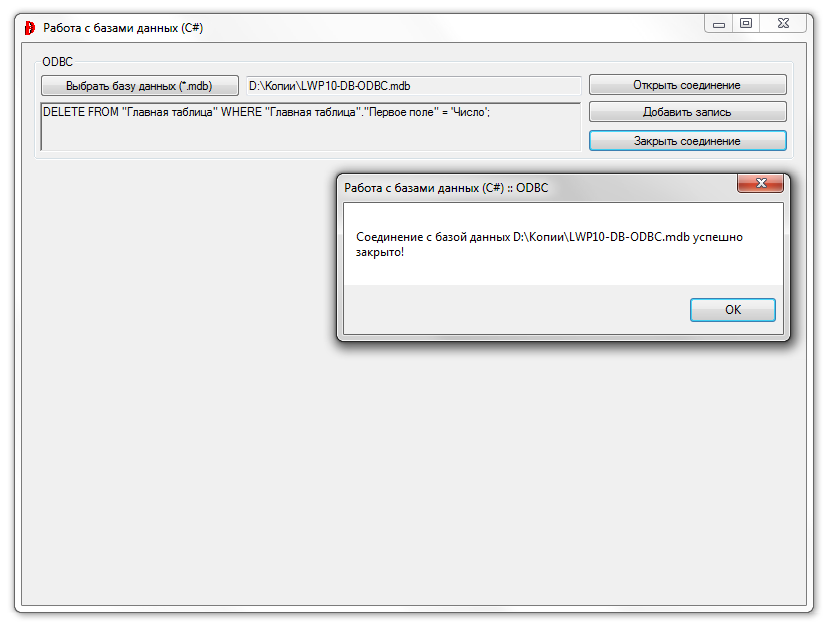


Рис. 3. 5. Окончательная работа блока: **ODBC**

**4. Модификация приложения Windows Forms: OLE**

Для соединения с базой данных Microsoft Office Access 2003 и ниже (файл **\*.mdb**) в C# следует использовать класс **OleDbConnection** со следующими параметрами соединения:

**Provider=Microsoft.Jet.OLEDB.4.0; Data Source=DataBaseFile**

Здесь DataBaseFile — абсолютный путь к файлу базы данных Access. «Провайдер» соединения должен иметь значение **Microsoft.Jet.OLEDB.4.0**.

Для соединения с базой данных Microsoft Office Access 2007 и выше (файл \*.accdb) в C# следует использовать класс **OleDbConnection** со следующими параметрами соединения:

**Provider=Microsoft.ACE.OLEDB.12.0; Data Source=DataBaseFile**

Здесь DataBaseFile — абсолютный путь к файлу базы данных Access. «Провайдер» соединения должен иметь значение **Microsoft.ACE.OLEDB.12.0** либо для версии Access 2010: **Microsoft.ACE.OLEDB.14.0**.

Пример:

String ConnetionString = null;

ConnetionStringO = "Provider=Microsoft.ACE.OLEDB.12.0;" +

@"Data Source=D:\Database.accdb";

ConnectionOLE1 = new OleDbConnection(ConnetionStringOLE1);

Для отправки SQL-запросов и чтения их результатов используются объекты **OleDbCommand** и **OleDbDataReader**.

Для выполнения запросов на вставку, изменение, или удаление данных из базы данных следует использовать метод класса *OleDbCommand*: **ExecuteNonQuery()**. Его вызов выполняет указанный в свойстве **CommandText** класса *OleDbCommand* запроc и возвращает int-число затронутых запросом полей.

Пример:

OleDbConnection Conn = new OleDbConnection(@"Provider=Microsoft.ACE.OLEDB.12.0; Data Source=Database.mdb"); // Инициализируем параметры соединения

Conn.Open(); // Открываем соединение с базой даннфх

OleDbCommand Cmd = new OleDbCommand(); // Создаём команду

Cmd.CommandText = "INSERT INTO Main\_Table VALUES (1,2,3)"; // Текст команды

int rowsAffected = Cmd.ExecuteNonQuery(); // Выполнение команды с возвратом в int

System.Windows.Forms.MessageBox.Show(rowsAffected.ToString()); // Вывод результата

Если база данных из примеры выше пустая, то int будет равно единице.

Во время выполнения команд вставки (SQL: **INSERT** **INTO**) можно использовать параметры, что может пригодиться, например, для добавления в базу данных файла изображения.

Пример:

OleDbCommand Сmd = new OleDbCommand("INSERT INTO Main\_Table (columns) VALUES (@param)", Сonn);

Сmd.Parameters.Add("@param", "abc[[''[]''kl'm");

Сmd.ExecuteNonQuery();

При каких-либо ошибках в синтаксисе, или структуре SQL-запроса, программа падает в **OS Loader Lock** (фактически прерывается работа), и не дает никакой информации о произошедшей ошибке. Чтобы этого избежать и получить довольно исчерпывающую информацию о произошедшей ошибке, следует выполнять запросы в try-catch блоках. В коде работы подобные приёмы будут использованы повсеместно.

Теперь приступим к формированию второй группы элементов и её функциональности. Для начала расставим элементы как показано на рисунке ниже:

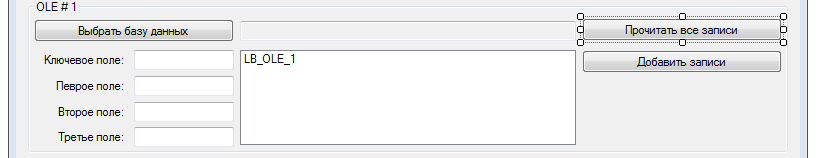


Рис. 4. 1. Расстановка элементов первой группы (**OLE # 1**)

Здесь представлены три кнопки *Button*, один *ListBox* (по центру), простой погашенный *TextBox*, как в предыдущей группе, и четыре TextBox’а для ввода с клавиатуры данных, которые будут добавлять в базу данных. Слева от каждого размещены по одному текстовому элементу: **Label**. Также с панели инструментов было добавлен ещё один *OpenFileDialog*-элемент.

Функциональность блока будет следующей. При нажатии на кнопку **Выбрать базу данных**, будет предложено выбрать одну из двух возможных типов базы: это база данных для старых версий Access (**\*.mdb**) и для новых (**\*.accdb**). После этого будет выдано сообщение об успехе или неудаче выбора с описанием причины ошибки в случае неудачи. Далее при нажатии **Прочитать все записи**, в центральный *ListBox* будут занесены все записи из базы данных. Ввод значений в *TextBox*’ы слева и нажатие кнопки Добавить записи сохранит в нашей базе новый записи. Будет использована та же самая база данных, что и для предыдущего блока: *LWP10-DB-ODBC.mdb*, а также копия этой же базы, сохранённая как файл **\*.accdb**, для версии Access 2007 или Access 2010. База: **LWP10-DB-OLE.accdb**.

По порядку распишем *Свойства* каждого элемента:

*Button:*

|  |  |
| --- | --- |
| **(Name)**: | B\_OLE\_1\_Search |
| **Text**: | Выбрать базу данных |
| **Size**: | 200; 23 |

*Button:*

|  |  |
| --- | --- |
| **(Name)**: | B\_OLE\_1\_Read |
| **Text**: | Прочитать все записи |
| **Size**: | 200; 23 |

*Button:*

|  |  |
| --- | --- |
| **(Name)**: | B\_OLE\_1\_Add |
| **Text**: | Добавить записи |
| **Size**: | 200; 23 |

*TextBox:*

|  |  |
| --- | --- |
| **(Name)**: | TB\_OLE\_1\_Path |
| **ReadOnly**: | True |

*ListBox:*

|  |  |
| --- | --- |
| **(Name)**: | LB\_OLE\_1 |

*GroupBox:*

|  |  |
| --- | --- |
| **(Name)**: | GB\_OLE\_1 |
| **Text**: | OLE # 1 |

*OpenFileDialog:*

|  |  |
| --- | --- |
| **(Name)**: | OFD\_OLE\_1 |
| **InitialDirectory**: | D:\ |
| **Filter** | База данных \*.mdb|\*.mdb|База данных \*.accdb|\*.accdb |

*TextBox:*

|  |  |
| --- | --- |
| **(Name)**: | TB\_OLE\_1\_1 |

*TextBox:*

|  |  |
| --- | --- |
| **(Name)**: | TB\_OLE\_1\_2 |

*TextBox:*

|  |  |
| --- | --- |
| **(Name)**: | TB\_OLE\_1\_3 |

*TextBox:*

|  |  |
| --- | --- |
| **(Name)**: | TB\_OLE\_1\_4 |

Откроем файл *LWP10Main.cs* и в самом начале добавим две ссылки:

using System.Data.OleDb; // OLE

using System.IO; // Для получения расширения файла базы данных

Найдём:

public partial class LWP10Main : Form

{

Double Counter = 4;

OdbcConnection ConnectionOBDC;

Добавим после:

String ConnetionStringOLE1 = null; // Переменная для сохранения данных соединения

OleDbConnection ConnectionOLE1; // Объект для открытия подключения к базе данных

String SQL\_OLE = null; // Переменная для поискового запроса

String SQL\_OLE\_ADD = null; // Переменная для добавления данных

Найдём:

public LWP10Main()

{

InitializeComponent();

TB\_ODBC\_Path.Text = "D:\\LWP10-DB-ODBC.mdb";

Добавим после:

TB\_OLE\_1\_Path.Text = "D:\\LWP10-DB-OLE.accdb";

B\_OLE\_1\_Read.Enabled = false;

B\_OLE\_1\_Add.Enabled = false;

TB\_OLE\_1\_1.Text = Counter.ToString();

SQL\_OLE = "SELECT \* FROM [Главная таблица]";

Событие *Click* кнопки **Выбрать базу данных**:

private void B\_OLE\_1\_Search\_Click(object sender, EventArgs e)

{

if (OFD\_OLE\_1.ShowDialog() == DialogResult.OK)

{

B\_OLE\_1\_Read.Enabled = true; // Активируем кнопку "Прочитать все записи"

B\_OLE\_1\_Add.Enabled = true;

//TB\_OLE\_1\_Path.Text = OFD\_OLE\_1.FileName;

Directory.CreateDirectory(Path.GetDirectoryName(OFD\_OLE\_1.FileName) + @"\Копии"); // Создаём директорию под изменённые БД

File.Copy(OFD\_OLE\_1.FileName, Path.GetDirectoryName(OFD\_OLE\_1.FileName) + @"\Копии\" + OFD\_OLE\_1.SafeFileName, true); // Копируем туда выбранную БД (перезаписываем, в случае обнаружения похожего файла)

if (Path.GetDirectoryName(OFD\_OLE\_1.FileName) == Directory.GetDirectoryRoot(OFD\_OLE\_1.FileName)) // Проверяем путь, если находимся в корневой директории диска, режем один слеш

TB\_OLE\_1\_Path.Text = Path.GetDirectoryName(OFD\_OLE\_1.FileName) + @"Копии\" + OFD\_OLE\_1.SafeFileName;

else

TB\_OLE\_1\_Path.Text = Path.GetDirectoryName(OFD\_OLE\_1.FileName) + @"\Копии\" + OFD\_OLE\_1.SafeFileName;

if (Path.GetExtension(OFD\_OLE\_1.FileName) == ".mdb") // Узнаём расширение файла выбранного в диалоге открытия (указываем полный путь) и сравниваем его с ".mdb"

{

ConnetionStringOLE1 = "Provider=Microsoft.Jet.OLEDB.4.0;" +

@"Data Source=" + TB\_OLE\_1\_Path.Text + "";

MessageBox.Show("Выбрана база данных " + TB\_OLE\_1\_Path.Text + " формата: \*" + Path.GetExtension(TB\_OLE\_1\_Path.Text) + "!", "Работа с базами данных (C#) :: OLE # 1");

}

if (Path.GetExtension(OFD\_OLE\_1.FileName) == ".accdb")

{

ConnetionStringOLE1 = "Provider=Microsoft.ACE.OLEDB.12.0;" +

@"Data Source=" + TB\_OLE\_1\_Path.Text + "";

MessageBox.Show("Выбрана база данных " + TB\_OLE\_1\_Path.Text + " формата: \*" + Path.GetExtension(TB\_OLE\_1\_Path.Text) + "!", "Работа с базами данных (C#) :: OLE # 1");

}

}

}

Событие *Click* кнопки **Показать все записи**:

private void B\_OLE\_1\_Read\_Click(object sender, EventArgs e)

{

LB\_OLE\_1.Items.Clear();

ConnectionOLE1 = new OleDbConnection(ConnetionStringOLE1);

try

{

ConnectionOLE1.Open(); // Открываем соединение

MessageBox.Show("Соединение с базой данных " + TB\_OLE\_1\_Path.Text + " успешно открыто!", "Работа с базами данных (C#) :: OLE # 1");

}

catch (Exception ex) // Ловим исключение и вытаскиваем ошибку через ex.Message

{

MessageBox.Show("Невозможно открыть соединение с базой данных " + TB\_OLE\_1\_Path.Text + " (" + ex.Message + ")!", "Работа с базами данных (C#) :: OLE # 1");

}

OleDbCommand Command = new OleDbCommand(SQL\_OLE, ConnectionOLE1); // Формируем SQL-команду для текущего подключения

OleDbDataReader DataReader = Command.ExecuteReader(); // Формируем объект для чтения данных из базы данных

LB\_OLE\_1.Items.Add(Command.CommandText); // Посылаем текст команды в ListBox

// Организуем циклический перебор полученных записей

while (DataReader.Read())

{

LB\_OLE\_1.Items.Add(DataReader["Ключевое поле"].ToString() + " | " + DataReader["Первое поле"].ToString() + " | " + DataReader["Второе поле"].ToString() + " | " + DataReader["Третье поле"].ToString());

}

// Закрываем потоки чтения и соединения

DataReader.Close();

ConnectionOLE1.Close();

}

Событие *Click* кнопки **Добавить записи**:

private void B\_OLE\_1\_Add\_Click(object sender, EventArgs e)

{

LB\_OLE\_1.Items.Clear(); // Очищаем ListBox перед использованием

ConnectionOLE1 = new OleDbConnection(ConnetionStringOLE1); // Передаём параметры объекту соединения

try

{

ConnectionOLE1.Open(); // Открываем соединение

MessageBox.Show("Соединение с базой данных " + TB\_OLE\_1\_Path.Text + " успешно открыто!", "Работа с базами данных (C#) :: OLE # 1");

}

catch (Exception ex)

{

MessageBox.Show("Невозможно открыть соединение с базой данных " + TB\_OLE\_1\_Path.Text + " (" + ex.Message + ")!", "Работа с базами данных (C#) :: OLE # 1");

}

SQL\_OLE\_ADD = "INSERT INTO [Главная таблица] VALUES('" + Counter.ToString() + "', '" + this.TB\_OLE\_1\_2.Text + "', '" + this.TB\_OLE\_1\_3.Text + "', '" + this.TB\_OLE\_1\_4.Text + "');";

OleDbCommand Command = new OleDbCommand(SQL\_OLE\_ADD, ConnectionOLE1);

try

{

Command.ExecuteNonQuery(); // Выполняем команду

}

catch (Exception ex)

{

MessageBox.Show("Невозможно выполнить команду с базой данных " + TB\_OLE\_1\_Path.Text + " (" + ex.Message + ")!", "Работа с базами данных (C#) :: OLE # 1");

}

LB\_OLE\_1.Items.Add(Command.CommandText); // Отправляем текст команды в ListBox

Counter++;

TB\_OLE\_1\_1.Text = Counter.ToString();

}

Компилируем приложение (**Debug**) и запускаем:

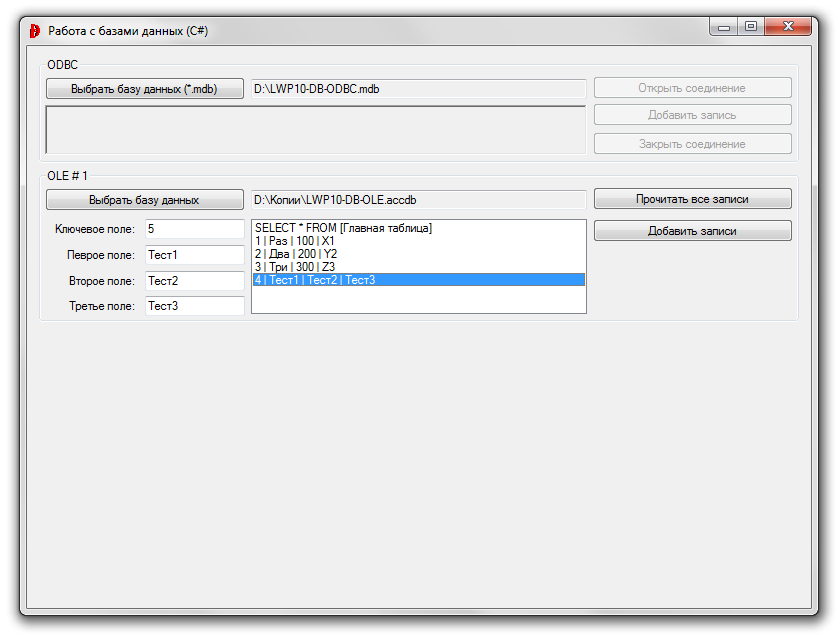
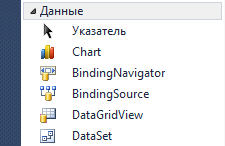
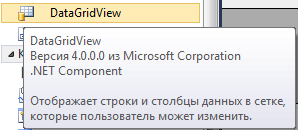


Рис. 4. 2. Окончательная работа блока: **OLE # 1**

Предположим, что работа в таком виде с базами данных нас не устраивает. Например, весьма неудобно для базы, где в таблице есть более сотни столбцов, делать эту самую сотню TextBox’ов. В этом случае, когда требуется наглядность данных в приложении и малые трудозатраты лучше нужно использовать элемент управления **DataGridView** и связанные с ним другие элементы, такие как например **DataSet**.



Оба элемента можно найти на панели элементов на вкладе **Данные**.



Собственно *DataGridView* полезен не только для работы с базами данных. Это просто элемент для предоставления данных в программе (в том числе и в визуальной форме).

Для демонстрации работы с этими элементами организуем третью группу элементов: OLE # 2. Приложение же будет выполнять тот же запрос, что и в предыдущем примере, но будет помещать результат в объект *DataSet*, который подключается к элементу управления *DataGridView*, а тот автоматически отображает все данные. Для сохранения данных в базе будет использоваться метод «двухсторонней привязки**3** данных», который позволит не только просматривать открывшуюся таблицу, но и вводить изменения в элемент *DataGridView*, добавляя новые строки, изменяя содержимое имеющихся строк и удаляя ненужные. Объекты класса **DataAdapter** способны выполнять как операцию **SELECT**, получая данные запроса из базы данных, так и команды **INSERT**, **UPDATE** и **DELETE**, изменяя содержимое таблицы базы данных.

Но прописывать данные команды самостоятельно надобности нет. Пространство имен **System.Data** содержит небольшой, но очень полезный класс **CommandBuilder**, который умеет создавать команды SQL и автоматически их выполнять.

**ПРИМЕЧАНИЕ № 3**: Подключение источника данных к визуальному элементу управления называется **привязкой**, или **связыванием данных**.

Теперь приступим к формированию третьей группы элементов и её функциональности. Для начала расставим элементы как показано на рисунке ниже:

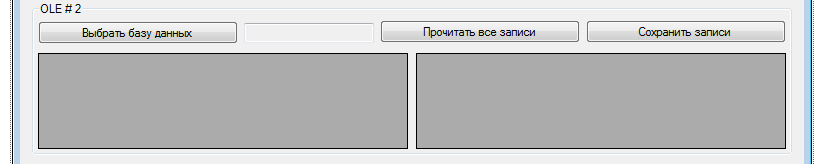
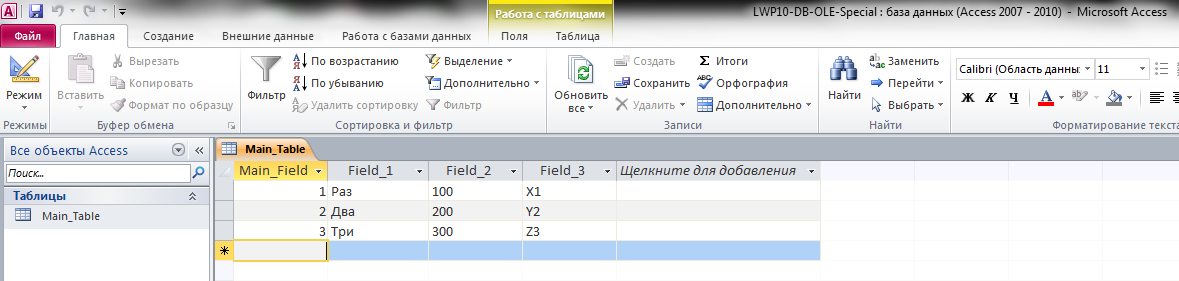


Рис. 4. 3. Расстановка элементов первой группы (**OLE # 2**)

Здесь представлены три кнопки *Button*, один, простой погашенный *TextBox*, как в предыдущей группе и два *DataGridView*. Также с панели инструментов было добавлен ещё один *OpenFileDialog*-элемент и один *DataSet*. Сразу оговоримся, для работы с добавлением данных через *DataGridView* база данных была переделана и название таблиц и столбцов были изменены (убраны символы кириллицы). Это связано с тем, что при работе с базой, в которой есть не английские символы в названиях таблиц (и столбцов) можно столкнуться с ошибками (не всегда). Сама же база осталась без изменений, Название: **LWP10-DB-OLE-Special.accdb**. Содержание таблицы **Main\_Table**:



Функциональность блока будет следующей. Выбираем базу данных как в предыдущих случаях и жмём на **Показать все записи**. Всплывающее сообщение с результатами, за которым идёт добавление всех данных базы в левый (по рисунку 4. 3) *DataGridView*. Правый же будет загружен сразу при старте приложения (база будет взята по статическому пути **D:\LWP10-DB-Special.accdb**. Вносим изменения в ячейки (в правом DataGridViewv) и жмём на **Сохранить записи**. Результат можно будет увидеть перезагрузив приложение, либо открыв базу через Access. Изменения ячеек в левом *DataGridView* не будут нигде сохранены.

По порядку распишем *Свойства* каждого элемента:

*Button:*

|  |  |
| --- | --- |
| **(Name)**: | B\_OLE\_2\_Search |
| **Text**: | Выбрать базу данных |
| **Size**: | 200; 23 |

*Button:*

|  |  |
| --- | --- |
| **(Name)**: | B\_OLE\_2\_Read |
| **Text**: | Прочитать все записи |
| **Size**: | 200; 23 |

*Button:*

|  |  |
| --- | --- |
| **(Name)**: | B\_OLE\_2\_Save |
| **Text**: | Сохранить записи |
| **Size**: | 200; 23 |

*TextBox:*

|  |  |
| --- | --- |
| **(Name)**: | TB\_OLE\_2\_Path |
| **ReadOnly**: | True |

*GroupBox:*

|  |  |
| --- | --- |
| **(Name)**: | GB\_OLE\_2 |
| **Text**: | OLE # 2 |

*OpenFileDialog:*

|  |  |
| --- | --- |
| **(Name)**: | OFD\_OLE\_2 |
| **InitialDirectory**: | D:\ |
| **Filter** | База данных \*.mdb|\*.mdb|База данных \*.accdb|\*.accdb |

*DataGridView:*

|  |  |
| --- | --- |
| **(Name)**: | DataGridViewOLE |

*DataGridView:*

|  |  |
| --- | --- |
| **(Name)**: | DataGridViewOLE\_S |

*DataSetw:*

|  |  |
| --- | --- |
| **(Name)**: | DataSetOLE |

Теперь перейдём к коду. Найдём:

String SQL\_OLE\_ADD = null; // Переменная для добавления данных

Добавим после:

String ConnetionStringOLE2 = null;

OleDbConnection ConnectionOLE2;

String ConnetionStringOLE\_S = null;

OleDbConnection ConnectionOLE\_S;

OleDbDataAdapter DataAdapter\_S;

Найдём:

SQL\_OLE = "SELECT \* FROM [Главная таблица]";

Добавим после:

TB\_OLE\_2\_Path.Text = "D:\\LWP10-DB-OLE.accdb";

B\_OLE\_2\_Read.Enabled = false;

DataGridViewOLE.DataMember = "Table"; // Указываем на тип подсписка для DataGridView

Событие *Click* кнопка **Выбор базы данных**:

private void B\_OLE\_2\_Search\_Click(object sender, EventArgs e)

{

if (OFD\_OLE\_2.ShowDialog() == DialogResult.OK)

{

B\_OLE\_2\_Read.Enabled = true;

B\_OLE\_2\_Save.Enabled = true;

//TB\_OLE\_2\_Path.Text = OFD\_OLE\_2.FileName;

Directory.CreateDirectory(Path.GetDirectoryName(OFD\_OLE\_2.FileName) + @"\Копии"); // Создаём директорию под изменённые БД

File.Copy(OFD\_OLE\_2.FileName, Path.GetDirectoryName(OFD\_OLE\_2.FileName) + @"\Копии\" + OFD\_OLE\_2.SafeFileName, true); // Копируем туда выбранную БД (перезаписываем, в случае обнаружения похожего файла)

if (Path.GetDirectoryName(OFD\_OLE\_2.FileName) == Directory.GetDirectoryRoot(OFD\_OLE\_2.FileName)) // Проверяем путь, если находимся в корневой директории диска, режем один слеш

TB\_OLE\_2\_Path.Text = Path.GetDirectoryName(OFD\_OLE\_2.FileName) + @"Копии\" + OFD\_OLE\_2.SafeFileName;

else

TB\_OLE\_2\_Path.Text = Path.GetDirectoryName(OFD\_OLE\_2.FileName) + @"\Копии\" + OFD\_OLE\_2.SafeFileName;

if (Path.GetExtension(OFD\_OLE\_2.FileName) == ".mdb")

{

ConnetionStringOLE2 = "Provider=Microsoft.Jet.OLEDB.4.0;" +

@"Data Source=" + TB\_OLE\_2\_Path.Text + "";

MessageBox.Show("Выбрана база данных " + TB\_OLE\_2\_Path.Text + " формата: \*" + Path.GetExtension(TB\_OLE\_2\_Path.Text) + "!", "Работа с базами данных (C#) :: OLE # 2");

}

if (Path.GetExtension(OFD\_OLE\_2.FileName) == ".accdb")

{

ConnetionStringOLE2 = "Provider=Microsoft.ACE.OLEDB.12.0;" +

@"Data Source=" + TB\_OLE\_2\_Path.Text + "";

MessageBox.Show("Выбрана база данных " + TB\_OLE\_2\_Path.Text + " формата: \*" + Path.GetExtension(TB\_OLE\_2\_Path.Text) + "!", "Работа с базами данных (C#) :: OLE # 2");

}

}

}

Событие *Click* кнопка **Показать все записи**:

private void B\_OLE\_2\_Read\_Click(object sender, EventArgs e)

{

DataSetOLE.Clear(); // Очищаем DataSetOLE перед повторным заполнением из базы данных

ConnectionOLE2 = new OleDbConnection(ConnetionStringOLE2);

try

{

ConnectionOLE2.Open(); // Открываем соединение

MessageBox.Show("Соединение с базой данных " + TB\_OLE\_2\_Path.Text + " успешно открыто!", "Работа с базами данных (C#) :: OLE # 2");

}

catch (Exception ex)

{

MessageBox.Show("Невозможно открыть соединение с базой данных " + TB\_OLE\_2\_Path.Text + " (" + ex.Message + ")!", "Работа с базами данных (C#) :: OLE # 2");

}

// Создаем объект DataAdapter и передаём ему данные запроса

OleDbDataAdapter DataAdapter = new OleDbDataAdapter(); // DataAdapter - посредник между базой данных и DataSet

DataAdapter.SelectCommand = new OleDbCommand(SQL\_OLE, ConnectionOLE2);

DataAdapter.Fill(DataSetOLE); // Данные из адаптера поступают в DataSet

DataGridViewOLE.DataSource = DataSetOLE; // Связываем данные с элементом DataGridView

// Закрываем соединение

ConnectionOLE2.Close();

}

Событие *Click* кнопка **Сохранить записи**:

private void B\_OLE\_2\_Save\_Click(object sender, EventArgs e)

{

try

{

DataAdapter\_S.Update((DataTable)DataGridViewOLE\_S.DataSource);

MessageBox.Show("Изменения в базе данных D:\\Копии\\LWP10-DB-OLE-Special.accdb успешно внесены!", "Работа с базами данных (C#) :: OLE # 2");

}

catch (Exception ex)

{

MessageBox.Show("Невозможно сохранить изменения в базе данных D:\\Копии\\LWP10-DB-OLE-Special.accdb (" + ex.Message + ")!", "Работа с базами данных (C#) :: OLE # 2");

}

}

Событие **Load** формы:

private void LWP10Main\_Load(object sender, EventArgs e)

{

Directory.CreateDirectory(@"D:\" + @"\Копии");

File.Copy(@"D:\LWP10-DB-OLE-Special.accdb", @"D:\" + @"\Копии\" + @"LWP10-DB-OLE-Special.accdb", true);

ConnetionStringOLE\_S = "Provider=Microsoft.ACE.OLEDB.12.0;" +

@"Data Source=D:\Копии\LWP10-DB-OLE-Special.accdb";

ConnectionOLE\_S = new OleDbConnection(ConnetionStringOLE\_S);

try

{

ConnectionOLE\_S.Open(); // Открываем соединение

MessageBox.Show("Соединение с базой данных D:\\Копии\\LWP10-DB-OLE-Special.accdb успешно открыто!", "Работа с базами данных (C#) :: OLE # 2");

}

catch (Exception ex)

{

MessageBox.Show("Невозможно открыть соединение с базой данных D:\\Копии\\LWP10-DB-OLE-Special.accdb (" + ex.Message + ")!", "Работа с базами данных (C#) :: OLE # 2");

}

DataTable DataTable\_S = new DataTable();

// Создаём команду

OleDbCommand Command = new OleDbCommand("SELECT \* FROM Main\_Table", ConnectionOLE\_S);

// Создаём адаптер DataAdapter\_S: посредник между базой данных и DataSet

DataAdapter\_S = new OleDbDataAdapter(Command);

// Создаём построитель команд

// Для адаптера становится доступной команда Update и другие команды

OleDbCommandBuilder CommandBuilder = new OleDbCommandBuilder(DataAdapter\_S);

// Данные из адаптера поступают в DataTable\_S

DataAdapter\_S.Fill(DataTable\_S);

// Связываем данные с элементом DataGridView

DataGridViewOLE\_S.DataSource = DataTable\_S;

// Закрываем соединение

ConnectionOLE\_S.Close();

}

Компилируем приложение (**Debug**) и запускаем:

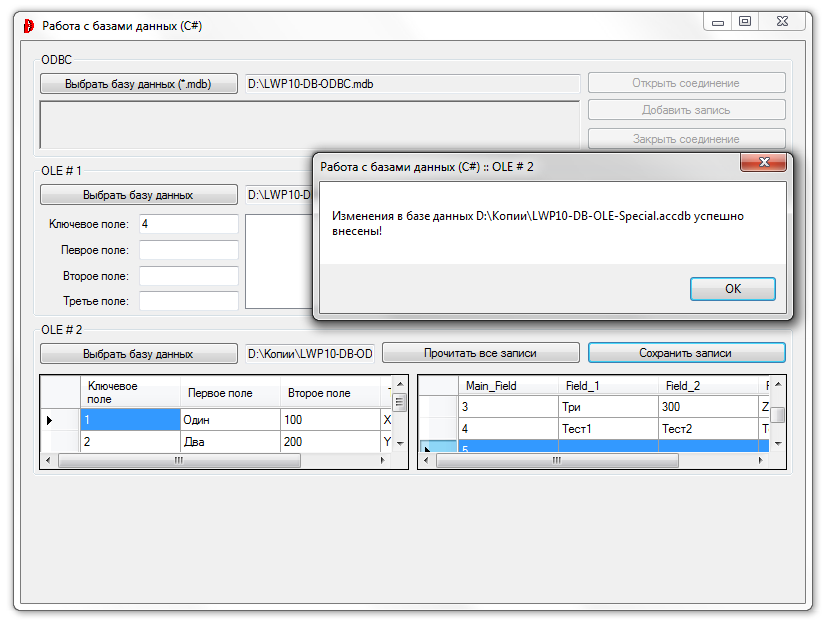


Рис. 4. 4. Окончательная работа блока: **OLE # 2**

**5. Модификация приложения Windows Forms: XML**

Немного общих слов об XML-формате в целом.

Язык XML (расширяемый язык разметки) предназначен для хранения структурированных данных. Данные, хранящиеся в формате XML, можно передавать между программами. Поскольку данные в XML структурированные, в некоторых случаях использование этого формата может заменить базы данных. Кроме этого, у XML есть еще много других полезных применений.

XML весьма похож на другой язык разметки **HTML**, но в *HTML* набор тегов фиксирован, и у каждого тега есть свое строго определённое назначение и правила написания. В XML можно определять собственные теги, но при этом все равно нужно соблюдать синтаксические правила языка разметки.

Типичный документ выглядит так:

<?xml version="1.0" encoding="UTF-8"?>

<recipe name="хлеб" preptime="5" cooktime="180">

<title>Простой хлеб</title>

<ingredient amount="3" unit="стакан">Мука</ingredient>

<ingredient amount="0.25" unit="грамм">Дрожжи</ingredient>

<ingredient amount="1.5" unit="стакан">Тёплая вода</ingredient>

<ingredient amount="1" unit="чайная ложка">Соль</ingredient>

<instructions>

<step>Смешать все ингредиенты и тщательно замесить.</step>

<step>Закрыть тканью и оставить на один час в тёплом помещении.</step>

<!-- <step>Почитать вчерашнюю газету.</step> - это сомнительный шаг... -->

<step>Замесить ещё раз, положить на противень и поставить в духовку.</step>

</instructions>

</recipe>

Главным является **объявление XML**:

<?xml version="1.0" encoding="UTF-8"?>

*Объявление* отвечает за кодировку документа (кодировка символов), наличие внешних зависимостей и версия документа. В версии **1.0** объявление можно не указывать. В версии **1.1** оно обязательно.

Важнейшее обязательное синтаксическое требование заключается в том, что документ имеет только один корневой элемент (англ. **root element**) (также иногда называемый элемент документа (англ. **document element**)). Это означает, что текст или другие данные всего документа должны быть расположены между единственным начальным корневым тегом и соответствующим ему конечным тегом.

Следующий простейший пример — правильно построенный документ XML:

<book>Это книга: "Книжечка"</book>

Комментарии доступны и здесь. Тэги внутри комментариев не обрабатываются:

<!-- Это комментарий -->

Остальная часть этого XML-документа состоит из вложенных элементов, некоторые из которых имеют атрибуты и содержимое. Элемент обычно состоит из открывающего и закрывающего тегов, обрамляющих текст и другие элементы. Открывающий тег состоит из имени элемента в угловых скобках, например, <step>, а закрывающий тег состоит из того же имени в угловых скобках, но перед именем ещё добавляется косая черта, например, </step>. Имена элементов, как и имена атрибутов, не могут содержать пробелы (иначе эти пробелы заменяются специальной последовательностью символов, что будет показано ниже), но могут быть на любом языке, поддерживаемом кодировкой XML-документа. Имя может начинаться с буквы, подчёркивания, двоеточия. Остальными символами имени могут быть те же символы, а также цифры, дефис, точка.

Содержимым элемента (англ. **content**) называется всё, что расположено между открывающим и закрывающим тегами, включая текст и другие (вложенные) элементы. Ниже приведён пример XML-элемента, который содержит открывающий тег, закрывающий тег и содержимое элемента:

<step>Замесить ещё раз, положить на противень и поставить в духовку.</step>

Кроме содержания у элемента могут быть атрибуты — пары имя-значение, добавляемые в открывающий тег после названия элемента. Значения атрибутов всегда заключаются в кавычки (одинарные или двойные), одно и то же имя атрибута не может встречаться дважды в одном элементе. Не рекомендуется использовать разные типы кавычек для значений атрибутов одного тега.

<ingredient amount="3" unit="стакан">Мука</ingredient>

Для данной работы вышеописанных свойств достаточно. Хотя XML обладает ещё целым рядом особенностей. Например, применение тэгов HTML, нетерпимость к перекрывающимся тэгам (один тэг, открывшийся раньше, закрывается раньше другого), наличие специальных символов HTML, пустых элементов и прочее.

Для начала добавим функциональности к уже имеющемся кода нашего приложения. Для этого в начале файла *LWP10Main.cs* добавим следующее:

using System.Xml; // XML

Найдём:

OleDbDataAdapter DataAdapter = new OleDbDataAdapter(); // DataAdapter - посредник между базой данных и DataSet

DataAdapter.SelectCommand = new OleDbCommand(SQL\_OLE, ConnectionOLE2);

DataAdapter.Fill(DataSetOLE); // Данные из адаптера поступают в DataSet

Добавим после:

DataSetOLE.DataSetName = "Главная таблица"; // Устанавливаем название корневого элемента XML-файла

DataSetOLE.WriteXml("D:\\LWP10-DB-XML.xml"); // Запишем данные таблицы в XML-файл

Скомпилируем приложение (**Debug**) и запустим. В третьем блоке элементов выберем базу данных и нажмём на *Прочитать все записи*. На диске D появится новый XML-файл: **LWP10-DB-XML.xml** с содержанием:



Рис. 5. 1. Содержание файла *LWP10-DB-XML.xml*

Как видно, пробелы в именах тэгов были заменены на символы «\_x0020\_». В остальном, документ точно повторяет структуру базы данных, из которой были выдраны данные.

Теперь создаём последнюю группу элементов. Она простая:

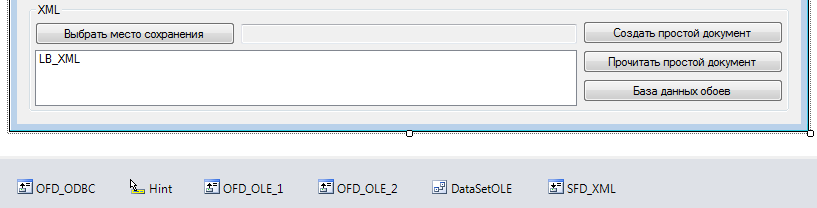


Рис. 5. 2. Расстановка элементов первой группы (**XML**)

Здесь у нас четыре кнопки, *TextBox* и большой *ListBox*. Также справа внизу элемент **SaveFileDialog**. *Свойства* элементов таковы:

*Button:*

|  |  |
| --- | --- |
| **(Name)**: | B\_XML\_Search |
| **Text**: | Выбрать место сохранения |
| **Size**: | 200; 23 |

*Button:*

|  |  |
| --- | --- |
| **(Name)**: | B\_XML\_Create |
| **Text**: | Создать простой документ |
| **Size**: | 200; 23 |

*Button:*

|  |  |
| --- | --- |
| **(Name)**: | B\_XML\_Read |
| **Text**: | Прочитать простой документ |
| **Size**: | 200; 23 |

*Button:*

|  |  |
| --- | --- |
| **(Name)**: | B\_XML\_DB |
| **Text**: | База данных обоев |
| **Size**: | 200; 23 |

*TextBox:*

|  |  |
| --- | --- |
| **(Name)**: | TB\_XML\_Path |
| **ReadOnly**: | True |

*ListBox:*

|  |  |
| --- | --- |
| **(Name)**: | LB\_XML |

*GroupBox:*

|  |  |
| --- | --- |
| **(Name)**: | GB\_XML |
| **Text**: | XML |

*SaveFileDialog:*

|  |  |
| --- | --- |
| **(Name)**: | SFD\_XML |
| **FileName**: | XML-Test |
| **InitialDirectory**: | D:\ |
| **Filter** | XML-файл|\*.xml |

Самая нижняя кнопка (**База данных обоев**) является кнопкой, открывающей новую форму, о чём пойдём немного ниже.

Событие *Click* кнопки **Выбрать место сохранения**:

private void B\_XML\_Search\_Click(object sender, EventArgs e)

{

SFD\_XML.ShowDialog();

TB\_XML\_Path.Text = SFD\_XML.FileName;

}

Событие *Click* кнопки **Создать простой документ**:

private void B\_XML\_Create\_Click(object sender, EventArgs e)

{

XmlWriterSettings SettingsXML = new XmlWriterSettings();

SettingsXML.Indent = true; // Включаем отступ для элементов XML-документа

SettingsXML.IndentChars = " "; // Задаём отступ (пробелами)

SettingsXML.NewLineChars = "\n"; // Задаём переход на новую строку

// Нужно ли опустить строку декларации формата XML документа

// Речь идёт о строке вида "<?xml version="1.0" encoding="utf-8"?>"

SettingsXML.OmitXmlDeclaration = false;

using (XmlWriter OutputXML = XmlWriter.Create(SFD\_XML.FileName, SettingsXML))

{

// Создали открывающийся тег

OutputXML.WriteStartElement("XML-Test");

// Добавляем атрибут для XML-Test

OutputXML.WriteAttributeString("Count\_Parameters", "4");

// Создаем элемент <имя>элемент</имя>

OutputXML.WriteElementString("Question", "Answer");

Random R = new Random();

OutputXML.WriteElementString("A", R.Next(0,1000).ToString());

OutputXML.WriteElementString("B", SQL\_OLE);

OutputXML.WriteElementString("C", TB\_XML\_Path.Text);

OutputXML.WriteStartElement("Names");

OutputXML.WriteStartElement("Name");

OutputXML.WriteAttributeString("Type", "Male");

OutputXML.WriteString("John");

OutputXML.WriteEndElement();

OutputXML.WriteStartElement("Name");

OutputXML.WriteAttributeString("Type", "Male");

OutputXML.WriteString("Teo");

OutputXML.WriteEndElement();

OutputXML.WriteStartElement("Name");

OutputXML.WriteAttributeString("Type", "Famale");

OutputXML.WriteString("Miana");

OutputXML.WriteEndElement();

// Закрываем XML-Test

OutputXML.WriteEndElement();

// Сбрасываем буфферизированные данные

OutputXML.Flush();

// Закрываем фаил, с которым связан output

OutputXML.Close();

MessageBox.Show("Документ " + SFD\_XML.FileName + " успешно создан!", "Работа с базами данных (C#) :: XML");

}

}

Событие *Click* кнопки **Прочитать простой документ**:

private void B\_XML\_Read\_Click(object sender, EventArgs e)

{

S = null;

LB\_XML.Items.Clear();

// Создаём экземпляр класса

XmlDocument InputXML = new XmlDocument();

// Загружаем XML-документ из файла

InputXML.Load(SFD\_XML.FileName);

// Загружаем XML-документ из строки

// InputXML.LoadXML(Path);

// Получаем всех детей корневого элемента

// InputXML.DocumentElement - корневой элемент

foreach (XmlNode Table in InputXML.DocumentElement.ChildNodes)

{

// Перебираем все атрибуты элемента

foreach (XmlAttribute A in Table.Attributes)

{

// A.Name - имя текущего атрибута

// A.Value - значение текущего атрибута

S = A.Name + ": " + A.Value;

}

// Перебираем всех детей текущего узла

foreach (XmlNode CN in Table.ChildNodes)

{

}

// Получаем текст хранящийся в текущем узле

S = S + Table.InnerText + "\n\t";

}

MessageBox.Show("Значения аттрибутов для элеметов и узлов файла "+ SFD\_XML.FileName +":\n\n\t" + S + "\nЗначения получены автоматическим перебором!", "Работа с базами данных (C#) :: XML");

// Вытаскиваем значения "руками"

XmlNodeList XMLTestAttributes = InputXML.SelectNodes("/XML-Test[@Count\_Parameters='4']");

foreach (XmlNode XN in XMLTestAttributes)

{

LB\_XML.Items.Add(XN.InnerText + " -- все значения"); // В ListBox получаем значения всех узлов

}

XmlNodeList Names = InputXML.SelectNodes("/XML-Test/Names/Name[@Type='Male']");

foreach (XmlNode XN in Names)

{

LB\_XML.Items.Add(XN.InnerText + " -- одно мужское имя"); // В ListBox получаем только два мужских имени по отдельности

}

XmlNode Question = InputXML.DocumentElement.SelectSingleNode("Question");

LB\_XML.Items.Add(Question.InnerXml);

XmlNode ANode = InputXML.DocumentElement.SelectSingleNode("A");

LB\_XML.Items.Add(ANode.InnerText);

XmlNode BNode = InputXML.DocumentElement.SelectSingleNode("B");

LB\_XML.Items.Add(BNode.InnerText);

XmlNode CNode = InputXML.DocumentElement.SelectSingleNode("C");

LB\_XML.Items.Add(CNode.InnerText);

XmlNode NamesNode = InputXML.DocumentElement.SelectSingleNode("Names");

LB\_XML.Items.Add(NamesNode.InnerText + " -- все имена"); // В Listbox получаем все имена

}

Компилируем приложение (**Debug**) и запускаем:

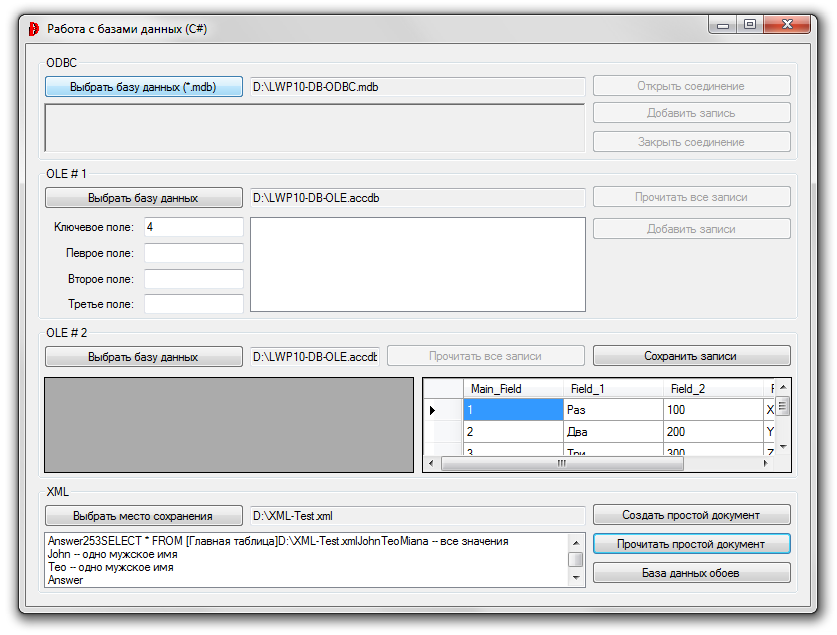
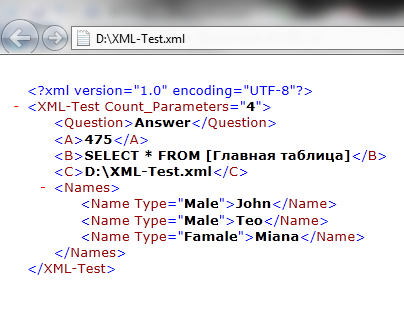


Рис. 5. 3. Окончательная работа блока: **XML**

Нажатие на кнопку *Создать простой документ*, создаёт по указанному пути файл (в данном случае по рисунку: **D:\XML-Test.xml**) следующего содержания:



При нажатии на кнопку *Прочитать простой документ*, происходит чтение всех записей в цикле (с выводом всплывающего окна), а затем вывод в ListBox. Выводятся все значения (верхняя строчка), затем только два мужских имени из блока **<Names>...</Names>**, затем все значения узлов **Question**, **A**, **B** и С и наконец все имена.

**Почему существует несколько классов для работы с одним и тем же (в частности в XML)?** На этот вопрос, есть два ответа: «на вкус и цвет все фломастеры разные» и «в зависимости от предоставляемой функциональности один класс может оказаться удобнее в той или иной ситуации».

Несколько причин, почему стоит использовать классы для генерации XML документа, а не создавать его вручную:

1. Практически нулевой риск создать неправильный XML документ.

2. Можно задать форматирование (отступы и прочее) XML документа (очень сильно улучшает читаемость).

Наиболее подходящим для формирования XML-документа «за раз» является класс **XmlWriter**. Плюсы данного класса:

1. Форматирование результирующего XML документа.

2. Создание открывающего и закрывающего тега в одну команду, причём для создания закрывающего тега не требуется знать имя открывающего.

Для форматирования выводимого XML документа в *XmlWriter* используется класс **XmlWriterSettings**.

Используя функцию **WriteElementString**,тэг не только открывается, в него записываются данные, но и тут же закрывается, поэтому добавить атрибут к такому элементу НЕЛЬЗЯ! Для решения данной проблемы нужно использовать **WriteStartElement**, **WriteAttributeString**, **WriteEndElement**.

Для **чтения XML-документа** предусмотрен класс **XMLReader**, но он крайне не удобен, потому что осуществляет последовательное считывание и не позволяет «прыгать через»/обращаться сразу к нужному элементу.

Чтение и редактирование удобно производить через класс **XmlDocument**.

Из чего состоит XML-документ:

1. Строка вида <?xml … ?>

2. Родительский элемент (единственный).

3. Дальше, внутри родительского элемента располагаются элементы-потомки (child’ы) и далее по иерархии.

**Теперь давайте рассмотрим как изменять уже имеющийся XML-документ?**

Для вставки элемента в иерархию XML-документа необходимо создать элемент типа **XmlNode** и задать его родителя, делается это следующим образом:

// Создаём node

// book - имя узла

XmlNode node = xmlDoc.CreateElement("book");

// Добавляем его в качестве ребенка

parentNode.AppendChild(node);

**Удаление узла?**

// Удаление узла

parentNode.RemoveChild(node);

**Работа с атрибутами**:

// Создаём новый атрибут

// genre - имя атрибута

XmlAttribute newAttr = doc.CreateAttribute("genre");

newAttr.Value = "novel"; // Задаём его значение

// Добавляем атрибут в коллекцию атрибутов элемента

node.SetNamedItem(newAttr);

// Удаляем атрибут

node.Attributes.RemoveNamedItem("genre");

// Изменяем значение атрибута

XmlAttributeCollection Attribs = node.Attributes;

XmlAttribute attr = (XmlAttribute)Attribs.GetNamedItem("genre");

attr.Value = "fiction";

Настало время разработать более-менее полезную функциональность. Используем возможности XML и сделаем на основе него полноценную базу данных.

Будем использовать загрузку данных из XML файла в *DataSet*, отметив, что после загрузки данных, доступ к данным ничем не отличается от работы с данными при их загрузке из таблиц БД.

Для XML справедливо: «Содержимым элемента (*content*) называется всё, что расположено между открывающим и закрывающим тегами. Это текст, вложенные элементы, комментарии и т.п.». Все спецификации XML документов подчёркивают, что именно текст, то, что является содержимым документа. Все средства для работы с XML документами, в том числе и методы класса *XmlDocument*, также ориентированы на текст (**CreateTextNode** и т.п.).

Из сказанного можно сделать вывод: хранить двоичные данные можно, только если они будут представлены как текстовые строки. Однако, прямое преобразование всего массива байт (двоичных данных) в текстовую строку не будет решением в силу того, что возможно присутствие в двоичном коде значений, совпадающих со служебными символами кодировки строк и значений, равных завершению строки. Как результат - мы сможем поместить в контент тэга только часть строки (до первого совпадения двоичного символа со значением окончания строки) при возможном отображении помещенного кода на нескольких строках XML файла для одного содержимого контента. Следовательно, строка должна формироваться и присваиваться значению контента как единое целое. Способы выполнить это могут быть разные, от посимвольного преобразования байт в значение char и последовательного добавления результата в содержимое контента (медленный способ), до использования **StringBuilder** (быстрый способ).

Наиболее часто хранение двоичных данных связано с задачей хранения рисунков (хотя приведенный ниже код будет равно хорошо работать и при использовании его для хранения в виде двоичных данных Web-страниц, Word или Excel-документов и прочее). Содержимое файла рисунка: массив **шестнадцатеричных символов**. Как следствие, и код представления рисунка в **content** опирается на шестнадцатеричное отображение строк.

**Так что же это будет за функциональность**? Приложение будет по указанному через диалог изображению создавать запись в базе данных (XML-файл), куда будет помещаться как сведения об изображении, так и само изображение в виде длинной строки символов. Также по номеру (ID рисунка) из базы можно будет восстановить это изображение и просмотреть его. Изображения можно будет только добавлять в базу, и просматривать добавленное уже из базы. Для этой функциональности у нас предусмотрена кнопка **База данных обоев**. А также заранее заготовленный файл **Wallpapper-DB.xml** с содержанием:

<?xml version="1.0" encoding="windows-1251"?>

<NewDataSet>

<wallpapper>

<id></id>

<name></name>

<pichash></pichash>

<picext></picext>

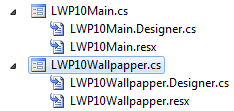
<size></size>

</wallpapper>

</NewDataSet>

Немного о тэге «**NewDataSet**». Поскольку мы будем работать с *DataSet*, то при сохранении содержимого *DataSet* в XML файл, данный тэг добавляется автоматически. Также этот тэг является параметром свойства *DataSetName*.

Первым делом нам нужна новая форма. Добавим её в наше приложение: **Проект** -> **Добавить новый элемент...** (**Ctrl**+**Shift**+**A**). В открывшемся окне ищем **Форма Windows Forms**, в поле **Имя** ниже вводим **LWP10Wallpapper.cs**:



Свойства формы:

|  |  |
| --- | --- |
| **(Name)**: | LWP10Wallpapper |
| **Text**: | Работа с базами данных (C#) :: База данных обоев |
| **Size**: | 500; 500 |

Расставим элементы как показано на рисунке ниже:

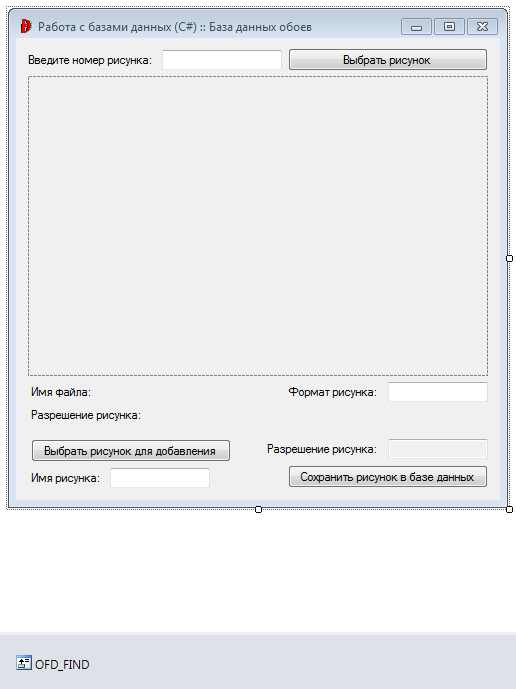


Рис. 5. 4. Расстановка элементов в новой форме

Основным элементом здесь является **PictureBox** (), который разместим в самом центре (пунктирный прямоугольник). Остальные элементы это три кнопки, четыре *TextBox*, два «именных» Label’а слева под рисунком (**Имя файла:** и **Разрешение рисунка:**).

*Свойства* элементов:

*Button:*

|  |  |
| --- | --- |
| **(Name)**: | B\_OPEN |
| **Text**: | Выбрать рисунок |
| **Size**: | 200; 23 |

*Button:*

|  |  |
| --- | --- |
| **(Name)**: | B\_SEARCH |
| **Text**: | Выбрать рисунок для добавления |
| **Size**: | 200; 23 |

*Button:*

|  |  |
| --- | --- |
| **(Name)**: | B\_SAVE |
| **Text**: | Сохранить рисунок в базе данных |
| **Size**: | 200; 23 |

*TextBox:*

|  |  |
| --- | --- |
| **(Name)**: | TB\_NUMBER |

*TextBox:*

|  |  |
| --- | --- |
| **(Name)**: | TB\_FORMAT |

*TextBox:*

|  |  |
| --- | --- |
| **(Name)**: | TB\_SIZE |
| **ReadOnly**: | True |

*TextBox:*

|  |  |
| --- | --- |
| **(Name)**: | TB\_NAME |

*PictureBox:*

|  |  |
| --- | --- |
| **(Name)**: | PB\_MAIN |

*OpenFileDialog:*

|  |  |
| --- | --- |
| **(Name)**: | OFD\_FIND |
| **InitialDirectory**: | D:\ |
| **Filter** | GIF-файлы|\*.gif|BMP-файлы|\*.bmp|JPEG-файлы|\*.jpg |

*Label:*

|  |  |
| --- | --- |
| **(Name)**: | L\_NAME |
| **Text**: | Имя файла: |

*Label:*

|  |  |
| --- | --- |
| **(Name)**: | L\_SIZE |
| **Text**: | Разрешение рисунка: |

Обработчик события *Click* для кнопки *База данных обоев* главной формы выглядит так:

private void B\_XML\_DB\_Click(object sender, EventArgs e)

{

LWP10Wallpapper DBWallpapper = new LWP10Wallpapper();

DBWallpapper.ShowDialog();

}

В начало файла *LWP10Wallpapper.cs* добавим следующие строчки:

using System.Xml;

using System.IO;

using System.Drawing.Imaging;

В этом же файле найдём:

public partial class LWP10Wallpapper : Form

{

Добавим после:

// Директория выполнения приложения

private string String\_Path = String.Empty;

XmlDocument XML;

// Классы для работв с XML-документом как с объектом базы данных

DataTable WallpapperDataTable = null;

DataSet WallpapperDataSet = null;

Событие **Load** нашей дочерней формы:

private void LWP10Wallpapper\_Load(object sender, EventArgs e)

{

//Path = Directory.GetCurrentDirectory(); // Текущая директория, из которой запущено приложения

String\_Path = @"D:\\";

L\_NAME.Text = "Имя рисунка: ";

L\_SIZE.Text = "Разрешение рисунка: ";

// Инициализируем объект StreamReader, считывающий символы из потока байтов в определённой кодировке

using (StreamReader SR = new StreamReader(String\_Path + @"\Wallpapper-DB.xml", System.Text.Encoding.UTF8))

{

WallpapperDataSet = new DataSet();

WallpapperDataSet.ReadXml(SR, XmlReadMode.Auto); // Считываем XML-схему и данные в DataSet

WallpapperDataTable = WallpapperDataSet.Tables[0];

}

}

Обратим внимание на параметр метода **ReadXml**: **XmlReadMode.Auto** - он позволит *DataSet* правильно создать схему с учётом значений мегатэгов, а также на значение **Encoding.UTF8**. Причина использования именно этой кодировки: *DataSet* по умолчанию будет сохранять данные в кодировке *UTF-8*.

В самом этом коде инициализируются основные пути к файлу базы данных, а также создаётся объект **StreamReader** (который занимается считыванием символов с файла в определённой кодировке, в данном случае **UTF-8**), после чего на основе XML-схемы формируется *DataSet* (столбцы таблицы).

Событие Click кнопки **Выбрать рисунок для добавления**:

private void B\_SEARCH\_Click(object sender, EventArgs e)

{

OFD\_FIND.InitialDirectory = String\_Path;

if (OFD\_FIND.ShowDialog() == DialogResult.OK)

{

PB\_MAIN.Image = Image.FromFile(OFD\_FIND.FileName);

TB\_FORMAT.Text = Path.GetExtension(OFD\_FIND.FileName);

TB\_NAME.Text = Path.GetFileNameWithoutExtension(OFD\_FIND.FileName);

TB\_SIZE.Text = PB\_MAIN.Image.Width.ToString() + "x" + PB\_MAIN.Image.Height.ToString();

}

}

Событие Click кнопки **Сохранить рисунок в базе данных**:

private void B\_SAVE\_Click(object sender, EventArgs e)

{

// Сохранять не будем - если что-то не ввели

if (PB\_MAIN.Image == null) return;

if (TB\_NAME.Text == "") return;

if (TB\_FORMAT.Text == "") return;

if (TB\_SIZE.Text == "") return;

// Строки для метода SELECT в XML-документе

DataRow[] datarows = null;

// Ищем максимальное ID в DataSet (в DataTable)

string s = string.Empty;

try

{

datarows = WallpapperDataTable.Select("id=max(id)");

s = datarows[0]["id"].ToString();

}

catch (Exception ex)

{

MessageBox.Show(ex.Message, "Работа с базами данных (C#) :: База данных обоев");

}

if (s == "" || s == string.Empty) // Если база данных пустая, то...

{

s = "0";

}

// Для формирования строки рисунка создаём StringBuilder

StringBuilder SB = new StringBuilder();

int i = int.Parse(s) + 1; // Если база пустая, то начинаем с 1, иначе, с максимального номера + 1

// Создаём новую строку для WallpapperDataSet

DataRow datarow = WallpapperDataSet.Tables[0].NewRow(); // Формируем DataRow на основе DataSet

// Присваиваем значения столбцам строки

datarow[0] = Convert.ToString(i);

datarow[1] = TB\_NAME.Text.Trim();

// Формируем строковое представление рисунка

using (MemoryStream MS = new MemoryStream())

{

PB\_MAIN.Image.Save(MS, ImageFormat.Gif); // Сохраняем изображение в потом MemoryStream, расширение \*.gif

byte[] b = new byte[MS.Length]; // 8-битное число (массив) длины потока в байтах

//memorystream.Read(b, 0, (int)memorystream.Length);

b = MS.GetBuffer(); // Присваиваем byte b массив байтов потока

s = string.Empty;

foreach (Byte zb in b)

{

int a = (int)zb;

SB.Append(a.ToString("X2")); // Формируем окончательную строку (путём добавления) из данных массива байтов в шестнадцатеричном виде (X2) (шестнадцатеричное представление каждого байта рисунка)

//value = 123456789;

//Console.WriteLine(value.ToString("X"));

// Выведет: 75BCD15

//Console.WriteLine(value.ToString("X2"));

// Выведет: 75BCD15

}

datarow[2] = Convert.ToString(SB); // Отправляем всю строку в столбец pichash нашей базы данных

}

datarow[3] = TB\_FORMAT.Text.Trim();

datarow[4] = TB\_SIZE.Text.Trim();

WallpapperDataSet.Tables[0].Rows.Add(datarow); // Формируем всю запись базы данных в DataSet

// Удаляем строку с пустыми значениями, которые при первоначальной

// загрузке были использованы для формирования схемы

if (i == 1)

{

WallpapperDataSet.Tables[0].DefaultView.AllowDelete = true;

WallpapperDataSet.Tables[0].DefaultView.Delete(0);

}

PB\_MAIN.Image = null;

TB\_SIZE.Text = "";

TB\_FORMAT.Text = "";

TB\_NAME.Text = "";

// Сохраняем данные

WallpapperDataSet.WriteXml(String\_Path + @"\Wallpapper-DB.xml", XmlWriteMode.WriteSchema);

WallpapperDataSet = new DataSet();

// Вновь загружаем сохраненные данные

WallpapperDataSet.ReadXml(String\_Path + @"\Wallpapper-DB.xml", XmlReadMode.Auto);

WallpapperDataTable = WallpapperDataSet.Tables[0];

}

Этот код выполняем основные функции по сохранению данных выбранного рисунка в базе данных. Вначале проверяется заполнение всех *TextBox*’ов (кроме верхнего). Если данные были введены и рисунок указан, далее проверяется наличие в базе ещё добавленных записей (по максимально числу в тэгах **<id>X</id>**). Если БД пустая, начинаем заполнение с номера 1. После чего инициализируется объект **StringBuilder** и производятся основные действия по переведению байтов рисунков в строчное представление для сохранение в базе. Данные *TextBox*’ов также заносятся в базу.

Теперь инициализируем извлечение данных из базы. Событие **KeyPress** для **TB\_SEARCH** (верхний *TextBox*):

private void TB\_NUMBER\_KeyPress(object sender, KeyPressEventArgs e)

{

// Введённые символы должны быть только цифрами, иначе ввода не будет (символ не введеётся)

if (!Char.IsDigit(e.KeyChar))

{

e.Handled = true;

}

}

Событие Click кнопки **Выбрать рисунок**:

private void B\_OPEN\_Click(object sender, EventArgs e)

{

if (TB\_NUMBER.Text.Trim() != "")

{

TB\_NAME.Text = "";

TB\_FORMAT.Text = "";

TB\_SIZE.Text = "";

PB\_MAIN.Image = null;

GetPicture(TB\_NUMBER.Text.Trim()); // Отправляем номер функции, которая вытащит из XML-документа все данные

}

}

И код функции для выбора по номеру, вводимому в *TB\_NUMBER*:

private void GetPicture(string X)

{

DataRow[] datarows = null;

try

{

datarows = WallpapperDataTable.Select("id=" + X); // Получаем все данные DataTable по ключу <id>X</id>

}

catch (Exception ex)

{

MessageBox.Show(ex.Message, "Работа с базами данных (C#) :: База данных обоев");

return;

}

if (datarows.Length > 0) // Не пусто ли?

{

foreach (DataRow datarow in datarows) // Перебираем все столбцы записи

{

L\_NAME.Text = "Имя файла: " + datarow["name"].ToString();

string s3 = datarow["pichash"].ToString();

L\_SIZE.Text = "Разрешение рисунка: " + datarow["size"].ToString();

using (MemoryStream MS = new MemoryStream())

{

for (int i = 0; i < s3.Length / 2; i++) // Обходим половину знаков строки из <pichash>s3</pichash> в верхнем цикле, макс. i = 249 если s3.Length = 500

{

if (i \* 2 + 1 < s3.Length) // Последнее условие: 249 \* 2 + 1 < 500

{

string s02 = Convert.ToString(s3[i \* 2]); // Чётные символы (посл.: 249 \* 2 = 498 символ, предпоследний в строке)

string s03 = Convert.ToString(s3[i \* 2 + 1]); // Нечётные символы (посл.: 249 \* 2 + 1 = 499 символ, последний в строке)

string s04 = s02 + s03; // Объединяем в одну строку (A + B)

int a = int.Parse(s04, System.Globalization.NumberStyles.HexNumber); // 32 разряда получаем из двух 16 разрядных чисел юникода (A + B)

MS.WriteByte(Convert.ToByte(a)); // Восстанавливает один байт за проход (из 32 разрядного представления знакового целого числа), всего 250 проходов по два символа за раз

}

}

PB\_MAIN.Image = Image.FromStream(MS); // Восстанавливаем рисунок (создаём рисунок из потока байтов)

}

}

}

}

Все данные для одного рисунка по номеру вытаскиваются этим кодом из файла базы данных в объект *DataTable*. Данные выводятся из файла через массив **DataRow[]** с непосредственным выбором по именам узлов. Из строки символов **<pichash>...</pichash>** формируются байты рисунка и затем передаются элементу *PictureBox*.

Готово. Можно компилировать и проверять работоспособность.

**6. Завершающая часть**

Компилируем приложение (**Release**) и запускаем. Результат работы показан ниже (Рис. 6. 1):

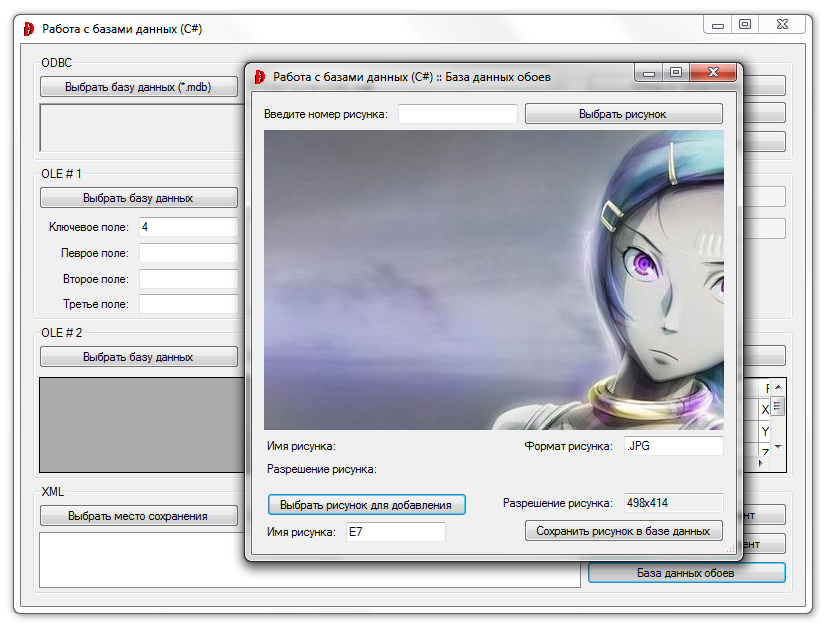


Рис. 6. 1. Модифицированное приложение *Windows Forms:* результат работы приложения по сохранению рисунка в базе данных (сохранение выбранного рисунка в базе)

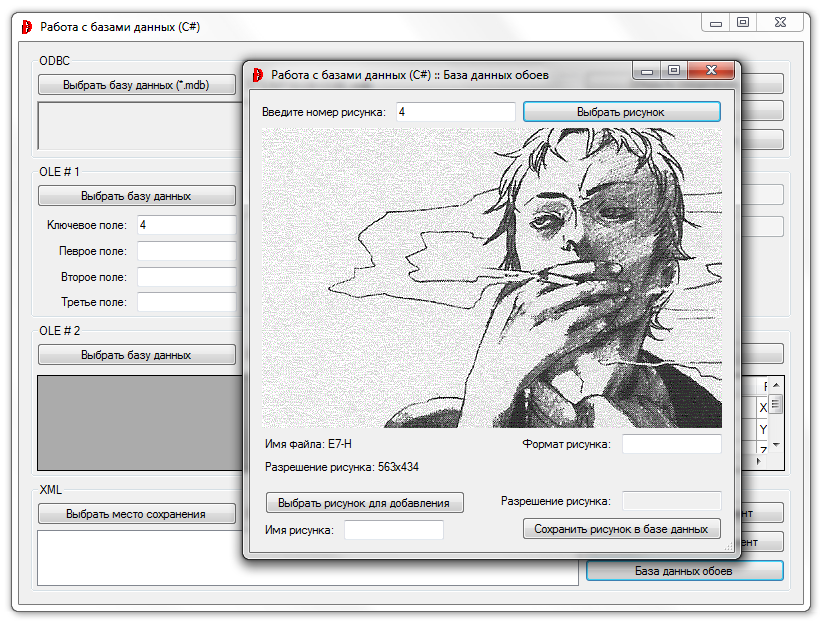


Рис. 6. 2. Модифицированное приложение *Windows Forms:* результат работы приложения по сохранению рисунка в базе данных (выбор рисунка из базы по номеру)

Содержание базы данных (файл *Wallpapper-DB.xml*):

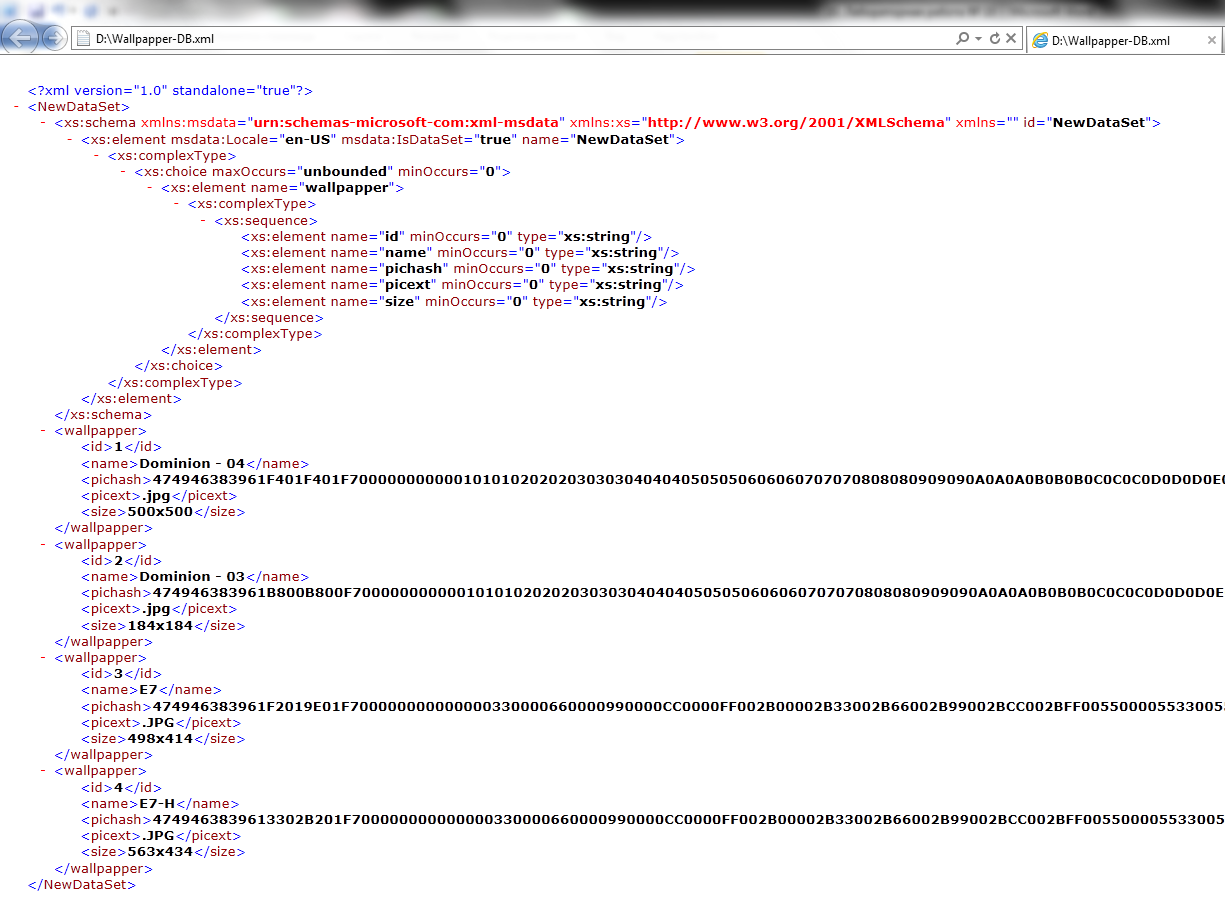


Рис. 6. 3. Содержимое файла *Wallpappaer-DB.xm*l: четыре рисунка, длинные строчки **<pichash>...</pichash>** содержат символы рисунков (окончания строк не видны)

Недостаток хранения данных в XML-файлах: длина строки, хотя это не значит, что при хранении в таблицах баз данных мы выиграем в объёме или скорости обработки (скорее наоборот проиграем за счёт сетевого трафика).

Если имена рисунков**4** будут на русском (символы кириллицы), они будут корректно сохранены в базе.

**ПРИМЕЧАНИЕ № 4**: Точно таким образом можно сохранять и восстанавливать любые данные (Web-страницы, Word или Excel-документы и прочее). Подобный способ хранения может быть полезен, для решения многих задач связанных с хранением и передачей информации, в том числе и конфиденциальной (например, после преобразования файла всю строку можно закодировать паролем через MD5-хэш).

Если по каким-то причинам русские символы не читаются, сделаем следующее:

Найдём:

using (StreamReader SR = new StreamReader(String\_Path + @"\Wallpapper-DB.xml", System.Text.Encoding.UTF8))

И заменим:

using (StreamReader SR = new StreamReader(String\_Path + @"\Wallpapper-DB.xml", System.Text.Encoding.Default))

Найдём:

// Сохраняем данные

WallpapperDataSet.WriteXml(String\_Path + @"\Wallpapper-DB.xml", XmlWriteMode.WriteSchema);

WallpapperDataSet = new DataSet();

// Вновь загружаем сохраненные данные

WallpapperDataSet.ReadXml(String\_Path + @"\Wallpapper-DB.xml", XmlReadMode.Auto);

WallpapperDataTable = WallpapperDataSet.Tables[0];

Заменим:

XML = new XmlDocument();

XML.InnerXml = WallpapperDataSet.GetXml();

XmlDeclaration XMLDeclaration = XML.CreateXmlDeclaration("1.0", "windows-1251", "yes");

XML.InsertBefore(XMLDeclaration, XML.DocumentElement);

XML.Save(String\_Path + @"\Wallpapper-DB.xml");

WallpapperDataSet = new DataSet();

WallpapperDataSet.ReadXml(String\_Path + @"\Wallpapper-DB.xml", XmlReadMode.Auto);

WallpapperDataTable = WallpapperDataSet.Tables[0];