Технологии доступа к данным. ADO .NET

Автор: доцент Асенчик О.Д.

Работа с внешними источниками данных

- ✔ Получение данных
- ✔ Представление данных в определенном формате для просмотра пользователем
- ✔ Обработку (редактирование) в соответствии с реализованными в программе алгоритмами
- ✔ Возврат обработанных данных в источник данных

Механизм доступа к внешнему источнику данных

ПО доступа к данным

Может быть реализовано как

- Программное окружение приложения, без которого приложение не сможет работать
- Набор драйверов и динамических библиотек
- Подпрограммы, интегрированные в само приложение
- Отдельный сервер, обслуживающий множество приложений

Технологии доступа к внешним источникам данных

- ✔ **ADO.NET** (ActiveX Data Object для .NET)
- ✔ **JDBC** (Java Database Connectivity)
- ✔ **BDE** (Borland Database Engine)
- ✔ **ADOdb**

ADO.NET

ADO.NET — набор классов, предоставляющих службы доступа к данным программисту, работающему на платформе .NET Framework.

ADO.NET имеет богатый набор компонентов для создания распределенных приложений, совместно использующих данные. Это неотъемлемая часть платформы .NET Framework, которая предоставляет доступ к:

- реляционным данным,
- XML-данным,
- данным приложений.

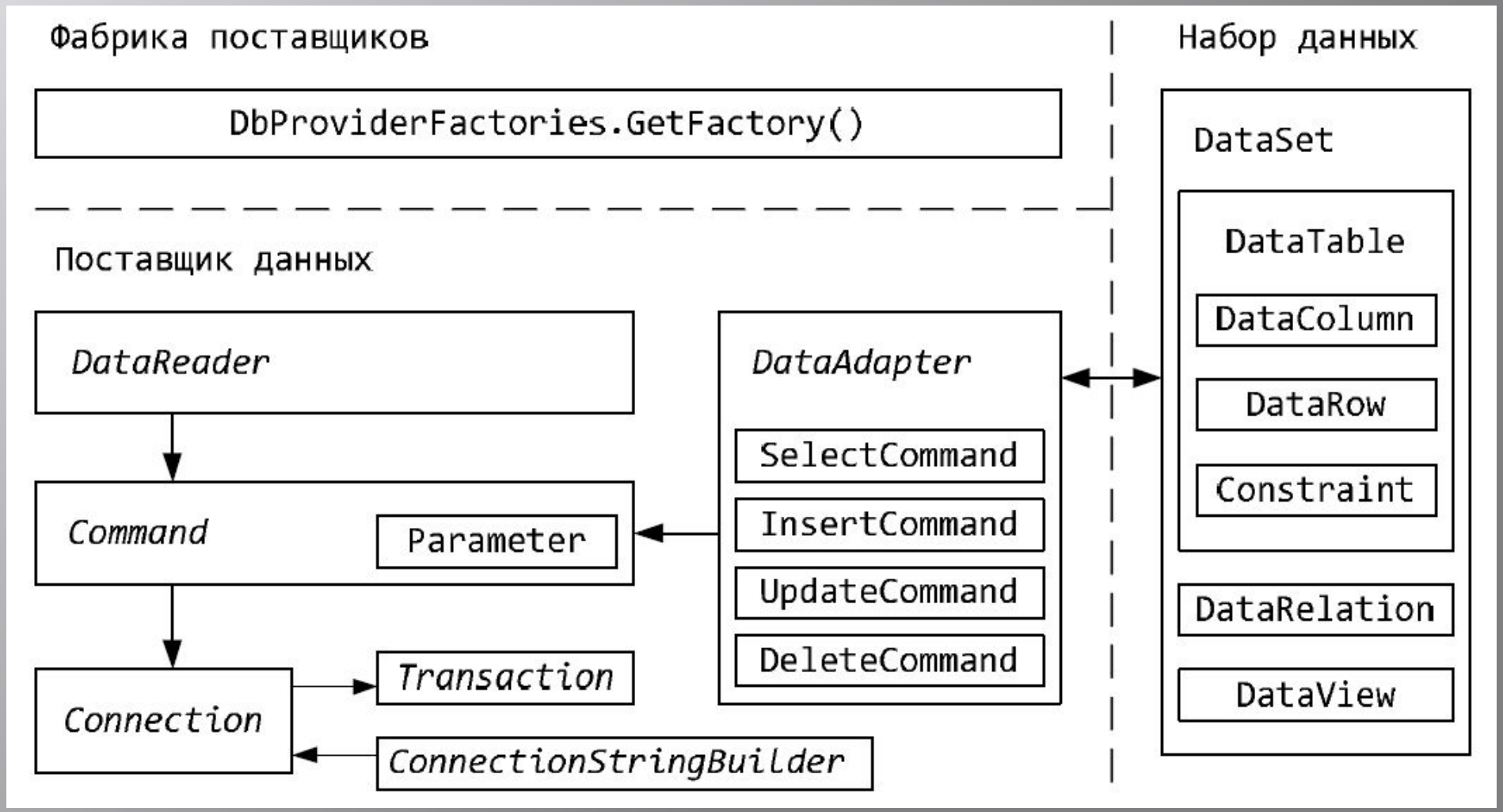
Фундаментальные классы ADO.NET

С точки зрения программиста, тело ADO.NET составляет базовая сборка с именем **System.Data.dll**. В этом двоичном файле находится значительное количество пространств имен, многие из которых представляют типы конкретного поставщика данных ADO.NET:

Фундаментальные классы ADO.NET

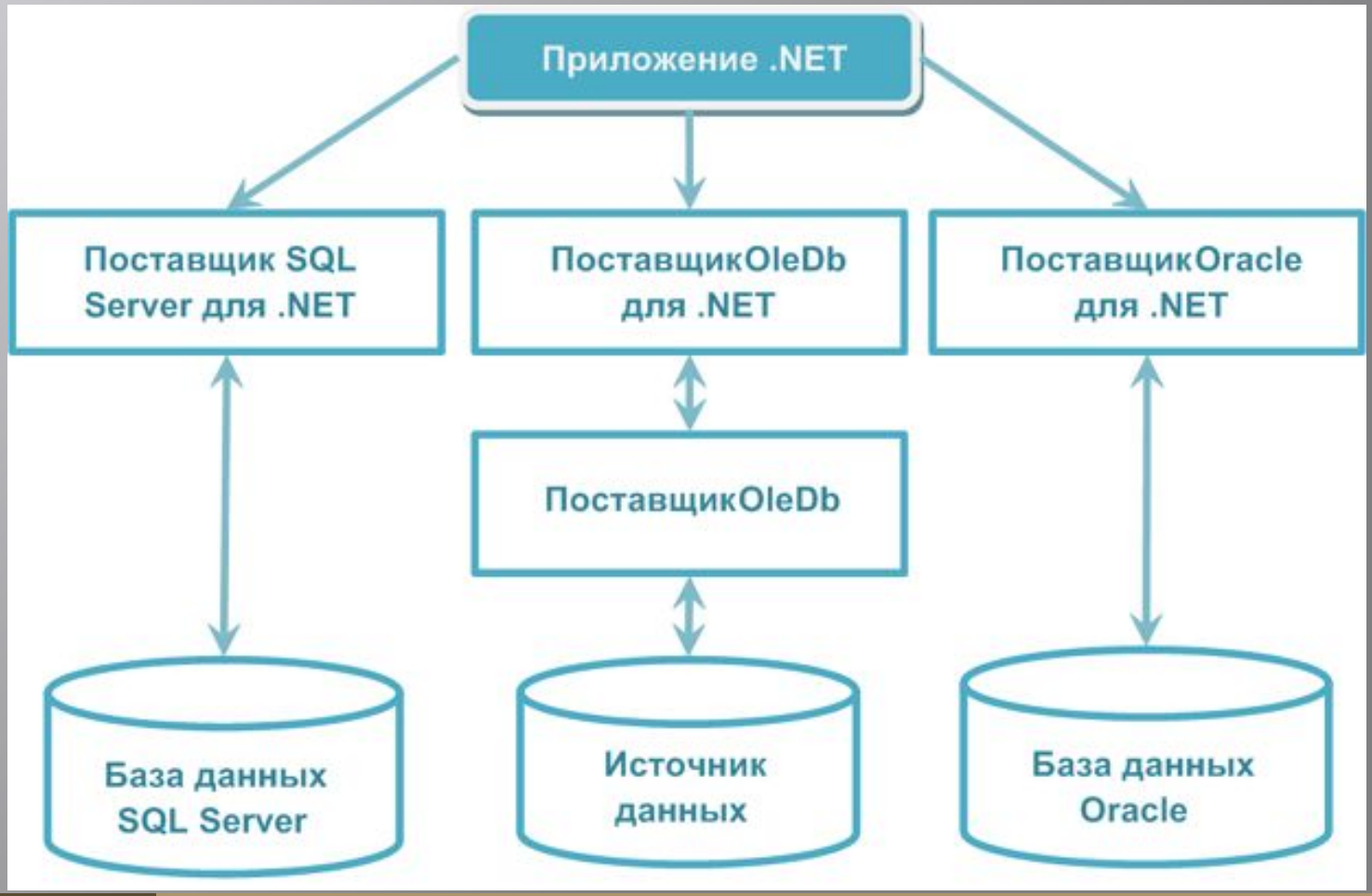
- **System.Data**- ключевые классы контейнеров данных, которые моделируют столбцы, отношения, таблицы, наборы данных, строки, представления и ограничения. Дополнительно содержит ключевые интерфейсы, которые реализованы объектами данных, основанными на соединениях
- **System.Data.Common** - базовые, наиболее абстрактные классы, которые реализуют некоторые из интерфейсов из System.Data и определяют ядро функциональности ADO.NET. Поставщики данных наследуются от этих классов (DbConnection, DbCommand и т.п.), создавая собственные специализированные версии
- **System.Data.OleDb** - классы, используемые для подключения к поставщику OLE DB, включая OleDbCommand, OleDbConnection и OleDbDataAdapter. Эти классы поддерживают большинство поставщиков OLE DB, но не те, что требуют интерфейсов OLE DB версии 2.5
- **System.Data.SqlClient** - классы, используемые для подключения к базе данных Microsoft SQL Server, в том числе SqlDbCommand, SqlDbConnection и SqlDbDataAdapter. **System.Data.OracleClient** - классы, необходимые для подключения к базе данных Oracle (версии 8.1.7 и выше), в том числе OracleCommand, OracleConnection и OracleDataAdapter.
- **System.Data.Odbc** - классы, необходимые для подключения к большинству драйверов ODBC, такие как OdbcCommand, OdbcConnection, OdbcDataReader и OdbcDataAdapter. Драйверы ODBC поставляются для всех видов источников данных и конфигурируются через значок Data Sources (Источники данных) панели управления
- **System.Data.SqlTypes** - структуры, соответствующие встроенным типам данных SQL Server. Эти классы не являются необходимыми, но предоставляют альтернативу применению стандартных типов данных .NET, требующих автоматического преобразования

Архитектура ADO.NET



Модель поставщиков

В основе ADO.NET лежит модель поставщиков, которая позволяет работать схожим образом с разными источниками данных:



Основные объекты поставщиков данных ADO.NET

Классы доступа к данным

- Connection
- Command
- DataReader
- DataAdapter

Классы для автономной обработки данных

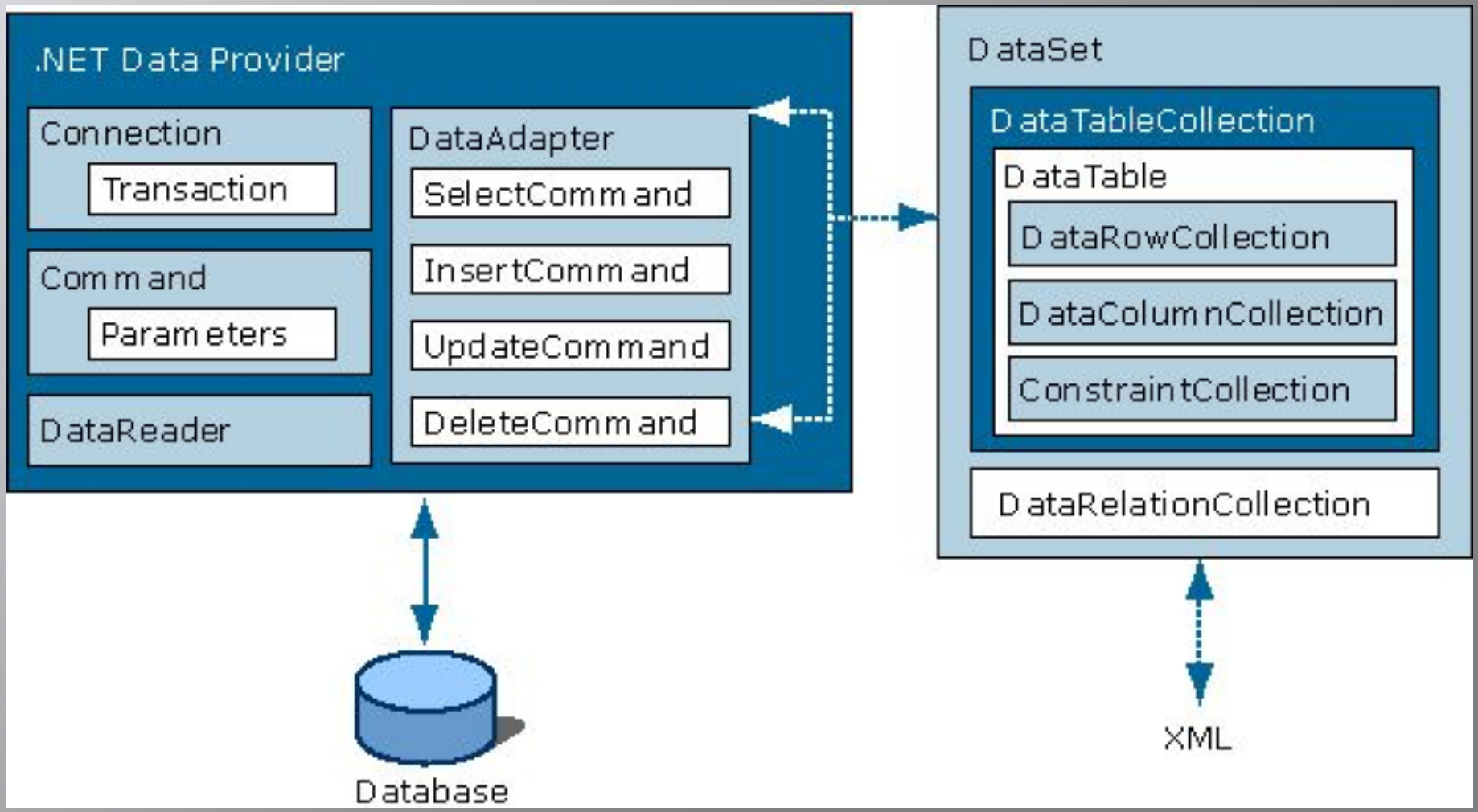
- DataSet
- DataTable
- DataColumn
- DataRelation
- etc.

Поставщики - структура

В состав поставщика входят следующие типы объектов:

- **Connection**. Позволяет подключаться к хранилищу данных и отключаться от него. Объекты подключения обеспечивают доступ к соответствующим объектам транзакций.
- **Command**. Представляет SQL-запрос или хранимую процедуру., предоставляют доступ к объекту чтения данных конкретного поставщика данных.
- **DataReader**. Этот объект предоставляет быстрый опережающий доступ только для чтения к данным, извлеченным по запросу.
- **DataAdapter**. Этот объект выполняет две задачи:
  - наполнение DataSet (автономная коллекция таблиц и отношений) информацией, извлеченной из источника данных.
  - применение изменений данных к источнику данных в соответствии с модификациями, произведенными в DataSet.

Поставщики - структура



Connection. Соединение с БД в ADO.NET

В основе подключения к базе лежит строка соединения:

```
Data Source=.\SQLEXPRESS;
AttachDbFilename=|DataDirectory|\PUBS.MDF;
Integrated Security=True;
```

В зависимости от источника данных строка соединения может включать такие элементы как:

- **Data Source** – адрес сервера.
- **Initial Catalog** – имя базы.
- **AttachDbFilename** – путь к файлу базы.
- **Integrated Security** – аутентификация Windows.
- **User Id** – идентификатор пользователя.
- **Password** – пароль.

https://www.connectionstrings.com/

Connection. Хранение строк соединения.

Строки лучше всего хранить в одном месте, чтобы из любого кода можно было бы просто получить доступ:

```
string connectionString =
WebConfigurationManager.ConnectionStrings["toplivoConnectionS
tring"].ConnectionString;
```

app.config (web.config) – специализированный файл конфигурации Windows (Web)- приложения, в нем есть раздел, предназначенный для хранения строк соединения:

```
<connectionStrings>
  <add … />
</connectionStrings>
```

Пример строки соединения:

```
 <add name="toplivoConnectionString"
connectionString="Data Source=.\sqlexpress;Initial
Catalog=toplivo;Integrated Security=True"
providerName="System.Data.SqlClient" />
```

Connection. Соединение с БД в ADO.NET

Параметры строки необходимо помнить и писать без ошибок, так как эти ошибки будут обнаружены только в момент подключения к источнику, но не при компиляции. В поставщиках данных имеется специальный класс, унаследованный от **DbConnectionStringBuilder**, который предназначен для построения правильных строк подключения. Типизированные свойства этого класса соответствуют отдельным параметрам строки подключения.

Пример демонстрирует использование класса **SqlConnectionStringBuilder**:

```
var builder = new SqlConnectionStringBuilder();
builder.DataSource = @"(.\sqlexpress";
builder.InitialCatalog ="toplivo";
builder.IntegratedSecurity = true;
Var  conn = new SqlConnection(builder.ConnectionString);
```

Connection. Соединение с БД в ADO.NET

Имея строку соединения можно подключится к базе данных, используется класс *Connection*:

```
SqlConnection conn = new SqlConnection(connString);
try
{
   conn.Open();
   …
}
finally
{
   conn.Close();
}
```

Серверы баз данных устанавливают лимит на количество одновременных подключений. Поэтому открытые соединения **следует закрывать** после использования. Для этого применяется метод ***Close()*** или метод ***Dispose()*** , являющийся реализацией интерфейса *IDisposable*. Использование ***using*** гарантирует вызов *connection.**Dispose()***

```
using( var conn = new SqlConnection(connString))
{
    conn.Open();
   // работа с соединением
}
```

Connection. Обработка ошибок соединения

Обычно при возникновении ошибки источника на клиенте генерируется исключительная ситуация особого типа, специфичного для каждого поставщика данных.

Например, для поставщика *SQL Server* это исключение типа ***SqlException***. В объекте *SqlException* доступно свойство *Errors* – набор объектов типа ***SqlError***. В этих объектах содержится дополнительная информация об ошибке:

```
try
        {
            // действия, которые могут вызвать ошибку
        }
        catch (SqlException ex)
        {
            string error = ex.Message + "\n";
            foreach(SqlError err in ex.Errors)
            {
                error += "Message: " + err.Message + "\n" +
                         "Level: " + err.Class + "\n" +
                         "Procedure: " + err.Procedure + "\n" +
                         "Line Number: " + err.LineNumber + "\n";
            }
        }
```

Класс Command

Класс **Command** позволяет выполнить SQL-оператор любого типа (*Create, Select, Update, Delete*):

```
SqlCommand com = new SqlCommand("Select * from Customer", conn);
```

Для выполнения оператора надо указать тип команды:

- ***Text*** - Команда будет выполнять прямой оператор SQL. Оператор SQL указывается в свойстве CommandText. Это — значение по умолчанию.
- ***StoredProcedure*** - Эта команда будет выполнять хранимую процедуру в источнике данных. Свойство CommandText представляет имя хранимой процедуры.
- ***TableDirect*** - Команда будет опрашивать все записи таблицы. CommandText — имя таблицы, из которой команда извлечет все записи.

Класс Command. Возможные значения перечисления CommandType

Значение	Пример SQL-запроса
Text (по умолчанию)	string select = "SELECT ContactName FROM Customers"; SqlCommand cmd = new SqlCommand(select, conn);

StoredProcedure	SqlCommand cmd = new SqlCommand("CustOrderHist", conn);
	cmd.CommandType = CommandType.StoredProcedure;
	cmd.Parameters.AddWithValue("@CustomerID", "QUICK");

TableDirect	OleDbCommand cmd = new OleDbCommand("Categories", conn); cmd.CommandType = CommandType.TableDirect;

Класс Command

После того как заданны все параметры для объекта Command, его можно выполнить одним из 3-х методов:

- ***ExecuteNonQuery()*** - Выполняет команды, отличные от SELECT, такие как SQL-операторы вставки, удаления или обновления записей. Возвращенное значение означает количество строк, обработанных командой. Также вы может использоваться для выполнения команд которые создают, изменяют и уничтожают объекты базы данных;
- ***ExecuteScalar()*** - Выполняет запрос SELECT и возвращает значение первого поля первой строки из набора строк, сгенерированного командой. Обычно применяется при выполнении агрегатной команды SELECT (вроде COUNT() или SUM() и др.);
- ***ExecuteReader()*** - Выполняет запрос SELECT и возвращает объект DataReader, который является оболочкой однонаправленного курсора, доступного только для чтения.

Пример выполнения команды:

```
SqlDataReader reader = com.ExecuteReader();
```

Класс Command. ExecuteNonQuery

Метод **ExecuteNonQuery()** обычно используется для операторов UPDATE, INSERT или DELETE, где единственным возвращаемым значением является количество обработанных строк. Однако метод может вернуть результаты, если осуществляется вызов хранимой процедуры с выходными параметрами.

```
string strSQL = "UPDATE Customers
SET LastName = 'Johnson'
WHERE LastName = 'Walton'";

SqlCommand myCommand = new
SqlCommand(strSQL,conn);
int i = myCommand.ExecuteNonQuery();
```

Класс Command. ExecuteScalar

Метод **ExecuteScalar** объекта Command выполняет запрос и возвращает первый столбец первой строки результирующего набора, возвращаемого запросом. Применяется для запросов, возвращающих одно значение. Такие запросы возникают, например, при использовании агрегатных функций COUNT, MIN, MAX.

```
string strSQL = "SELECT COUNT (*) FROM Inventory";
SqlCommand myCommand = new SqlCommand(strSQL, conn);
int countInv = myCommand.ExecuteScalar();
Console.WriteLine("Количество записей" +
Convert.ToString(countInv));
```

Класс Command. ExecuteReader

Метод **ExecuteReader()** выполняет команду и возвращает типизированный объект-читатель данных, в зависимости от используемого поставщика. Возвращенный объект может применяться для итерации по возвращенным записям.

```
string strSQL = "SELECT * FROM Inventory";
SqlCommand myCommand = new SqlCommand(strSQL,conn);
SqlDataReader dr = myCommand.ExecuteReader();
while (dr.Read())
     Console.WriteLine("ID: {0} Car Pet Name:
{1}",dr[0],dr[3]);
```

Класс Command. Параметризированные запросы.

```
protected void cmdGetRecords_Click(object sender, EventArgs e)
{
    string connectionString = WebConfigurationManager.ConnectionStrings["Northwind"].ConnectionString;
    SqlConnection con = new SqlConnection(connectionString);

    // Получить заказы по идентификатору заказчика
    string sql =
      "SELECT Orders.CustomerID, Orders.OrderID, COUNT(UnitPrice) AS Items, " +
      "SUM(UnitPrice * Quantity) AS Total FROM Orders " +
      "INNER JOIN [Order Details] " +
      "ON Orders.OrderID = [Order Details].OrderID " +
      "WHERE Orders.CustomerID = '" + txtID.Text + "' " +
      "GROUP BY Orders.OrderID, Orders.CustomerID";

    SqlCommand cmd = new SqlCommand(sql, conn);

    conn.Open();
    SqlDataReader reader = cmd.ExecuteReader();
    GridView1.DataSource = reader;
    GridView1.DataBind();
    reader.Close();
    conn.Close();
}
```

CustomerID	OrderID	Items	Total
ALFKI	10643	3	1086,0000
ALFKI	10692	1	878,0000
ALFKI	10702	2	330,0000
ALFKI	10835	2	851,0000
ALFKI	10952	2	491,2000
ALFKI	11011	2	960,0000

Класс Command. Параметризированные запросы.

Атака внедрением:

Вот что пользователь может ввести в текстовом поле, чтобы осуществить более изощренную атаку внедрением SQL, удалив все строки в таблице Customers:

```
ALFKI'); DELETE * FROM Customers—
```

; - граница начала нового оператора

-- - комментирует оставшуюся часть оператора.

Что бы избежать этой опасности, нужно использовать **параметры**.

Класс Command. Параметризованная команда.

Параметризованная команда — это просто команда, которая использует символы-заполнители в тексте SQL. Заполнитель указывает место для динамически применяемых значений, которые затем пересылаются через коллекцию **Parameters** объекту Command.

Например, следующий оператор SQL:

```
SELECT * FROM Customers WHERE CustomerID = 'ALFKI'
```

должен стать чем-то вроде:

```
SELECT * FROM Customers WHERE CustomerID = @CustID
```

@CustID значение будет считаться единым целым, независимо от того, что ввел пользователь.

Класс Command. Parameters.

У класса Command есть коллекция **Parameters**, через которую реализуется передача параметризированных команд на сервер:

```
SqlCommand com = new SqlCommand("Select CityName from
City where CountryID = (select CountryID from Country
where CountryName = @Name)", conn);

com.Parameters.AddWithValue("@Name", CountryName);
```

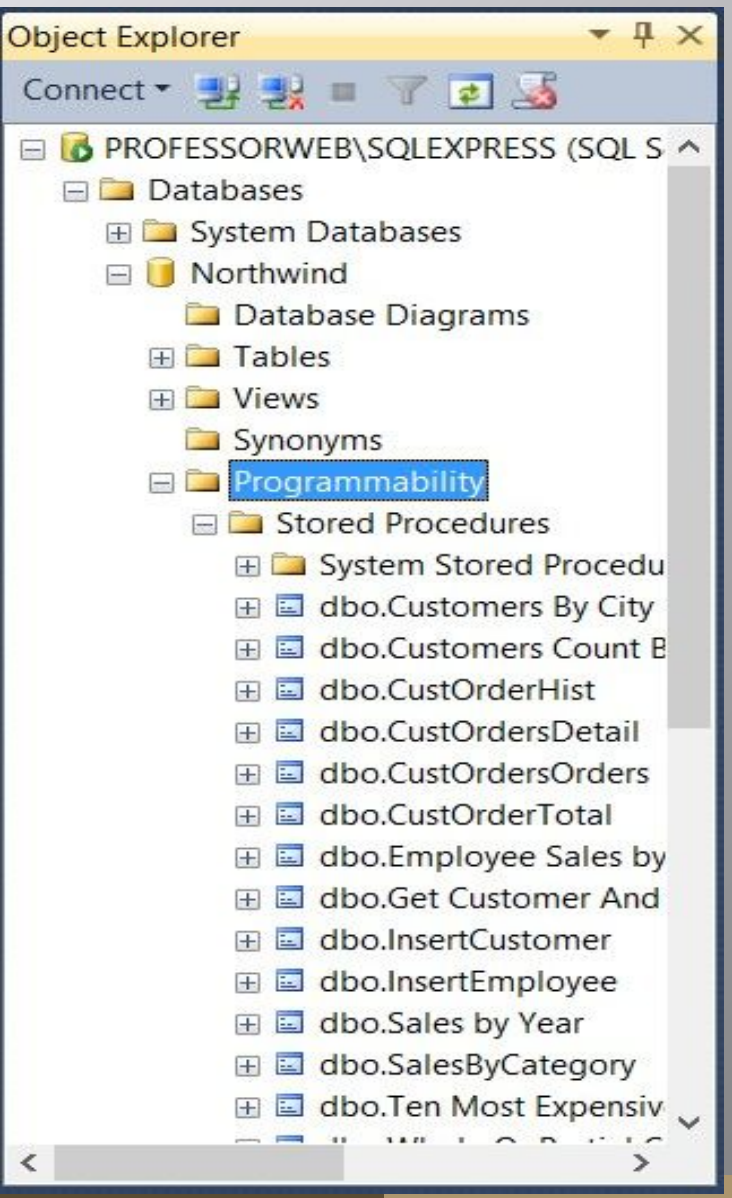
Такой оператор гарантированно избежит SQL инъекции.

Хранимые процедуры в ADO.NET

Выполнение в базе данных хранимых процедур вместо отдельных операторов SQL дает пользователю следующие преимущества:

- необходимые операторы уже содержатся в базе данных;
- все они прошли этап синтаксического анализа и находятся в исполняемом формате; перед выполнением хранимой процедуры SQL Server генерирует для нее план исполнения, выполняет ее оптимизацию и компиляцию;
- хранимые процедуры поддерживают модульное программирование, так как позволяют разбивать большие задачи на самостоятельные, более мелкие и удобные в управлении части;
- хранимые процедуры могут вызывать другие хранимые процедуры и функции;
- хранимые процедуры могут быть вызваны из прикладных программ других типов;
- как правило, хранимые процедуры выполняются быстрее, чем последовательность отдельных операторов;
- хранимые процедуры проще использовать: они могут состоять из десятков и сотен команд, но для их запуска достаточно указать всего лишь имя нужной хранимой процедуры. Это позволяет уменьшить размер запроса, посылаемого от клиента на сервер, а значит, и нагрузку на сеть.

Хранимые процедуры в ADO.NET



Рассмотрим пример. Создадим процедуру:

```
CREATE PROCEDURE dbo.AddCustomer

@CityName nvarchar(255),
@CustomerName nvarchar(255)

AS
BEGIN
insert into Customer
(CustomerName,CityID)
values (@CustomerName,
select CityID
from City
where CityName = @CityName));
END
```

Хранимые процедуры в ADO.NET

Вызов процедуры в клиентской программе:

```
SqlCommand com = new SqlCommand("AddCustomer", conn);
com.CommandType = CommandType.StoredProcedure;

com.Parameters.AddWithValue("@CityName", “Гомель”);
com.Parameters.AddWithValue("@CustomerName", “Ковбеня Э.
М.”);

try
{
  conn.Open();
  com.ExecuteNonQuery();
}
catch (Exception e)
{}
finally
{
  conn.Close();
}
```

Хранимые процедуры в ADO.NET. Выходной параметр.

Некоторые процедуры могут иметь выходной (OUTPUT) параметр:

```
CREATE PROCEDURE InsertEmployee
@TitleOfCourtesy varchar(25),
@LastName varchar(20),
@FirstName varchar(10),
@EmployeeID int OUTPUT
AS
INSERT INTO Employees
(TitleOfCourtesy, LastName, FirstName, HireDate)
VALUES (@TitleOfCourtesy, @LastName, @FirstName,
GETDATE());
SET @EmployeeID = @@IDENTITY
```

Хранимые процедуры в ADO.NET. Выходной параметр.

Прочитать выходной параметр достаточно просто:

```
cmd.Parameters.Add(new SqlParameter("@EmployeeID", SqlDbType.Int, 4));
cmd.Parameters["@EmployeeID"].Direction = ParameterDirection.Output;

try
{
  conn.Open();
  cmd.ExecuteNonQuery();
  int empID = (int)cmd.Parameters["@EmployeeID"].Value;
}
finally
{
  conn.Close();
}
```

Класс DataReader.

DataReader позволяет читать данные, возвращенные командой SELECT, по одной строке за раз, в однонаправленном, доступном только для чтения потоке

```
SqlConnection conn = new SqlConnection(connString);
SqlCommand com = new SqlCommand("Select CountryName from Country", con);
List<String> retList = new List<String>();
{
  conn.Open();
  SqlDataReader reader = com.ExecuteReader();
  while (reader.Read())
    {
      retList.Add(reader.GetValue(0).ToString());
    }
}
catch (Exception e)
{}
finally
{
  conn.Close();
}
```

Класс DataReader. Методы.

DataReader представляет наиболее быстрый способ осмысленного доступа к данным и имеет следующие методы:

- ***Read()*** - Перемещает курсор строки на следующую строку в потоке. Метод **Read()** возвращает true, если существует следующая строка для прочтения:

```
while (reader.Read())
{ }
```

- ***GetValue()*** **-** Возвращает значение, сохраненное в поле с указанным именем столбцы или индексом, внутри текущей выбранной строки. Тип возвращенного значения — ближайший тип .NET, наиболее соответствующий “родному” значению, хранимому в источнике данных.
- ***GetValues()*** - Сохраняет значения текущей строки в массиве. Количество сохраняемых полей зависит от размера массива, переданного этому методу. Можно использовать свойство DataReader.FieldCount для определения действительного числа полей в строке, и использовать эту информацию для создания массива нужного размера

Класс DataReader. Методы.

- ***GetInt32(), GetChar(), GetDateTime()*** и т.д. - Эти методы возвращают значение поля с указанным индексом в текущей строке, причем тип данных специфицируется именем метода. Эти методы не поддерживают типов, допускающих null-значения.
- ***NextResult()*** - Если команда, которая сгенерировала DataReader, возвратила более одного набора строк, этот метод перемещает указатель на следующий набор строк и устанавливает его непосредственно перед первой строкой.

```
while (reader.Read())
{
  retList.Add(reader.GetValue(0).ToString());
}
```

Транзакции

- Транзакция — это набор операций в базе данных, которые должны быть либо все выполнены, либо все не выполнены

Транзакции. Пример

```
 // Выборка имени по идентификатору клиента
string fName = string.Empty;
string lName = string.Empty;
SqlCommand cmdSelect =
new SqlCommand(string.Format("Select * from Customers where CustID =
{0}", custId), cn);
using (SqlDataReader dr = cmdSelect.ExecuteReader())
{
    if (dr.HasRows)
    {
        dr.Read();
        fName = (string)dr["FirstName"];
        lName = (string)dr["LastName"];
    }
    else return;
}
```

Транзакции. Пример

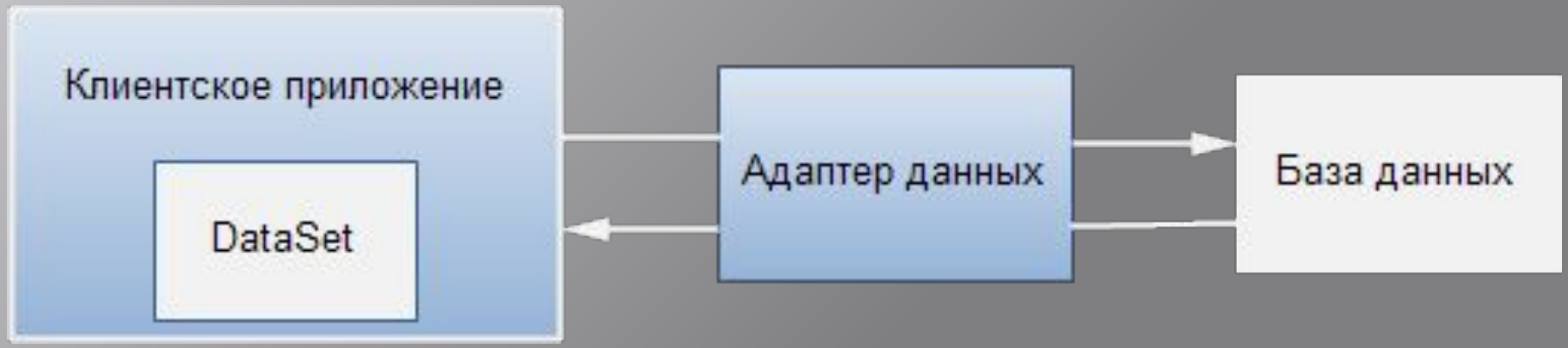
```
// Создание объектов команд для каждого шага операции.
SqlCommand cmdRemove = new SqlCommand(
string.Format("Delete from Customers where CustID = {0}",
custId), cn);

SqlCommand cmdInsert = new SqlCommand(string.Format("Insert
Into   CreditRisks" + "(CustID, FirstName, LastName) Values" +
       "({0}, '{1}', '{2}')", custId, fName, lName), cn);
```

Транзакции. Пример

```
SqlTransaction tx = null;
try
{
    tx = cn.BeginTransaction();
    // Включение команд в транзакцию
    cmdInsert.Transaction = tx;
    cmdRemove.Transaction = tx;
    // Выполнение команд.
    cmdInsert.ExecuteNonQuery();
    cmdRemove.ExecuteNonQuery();
    tx.Commit();
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
    // При возникновении любой ошибки выполняется откат транзакции.
    tx.Rollback();
}
```

Отсоединенные наборы данных в ADO.NET

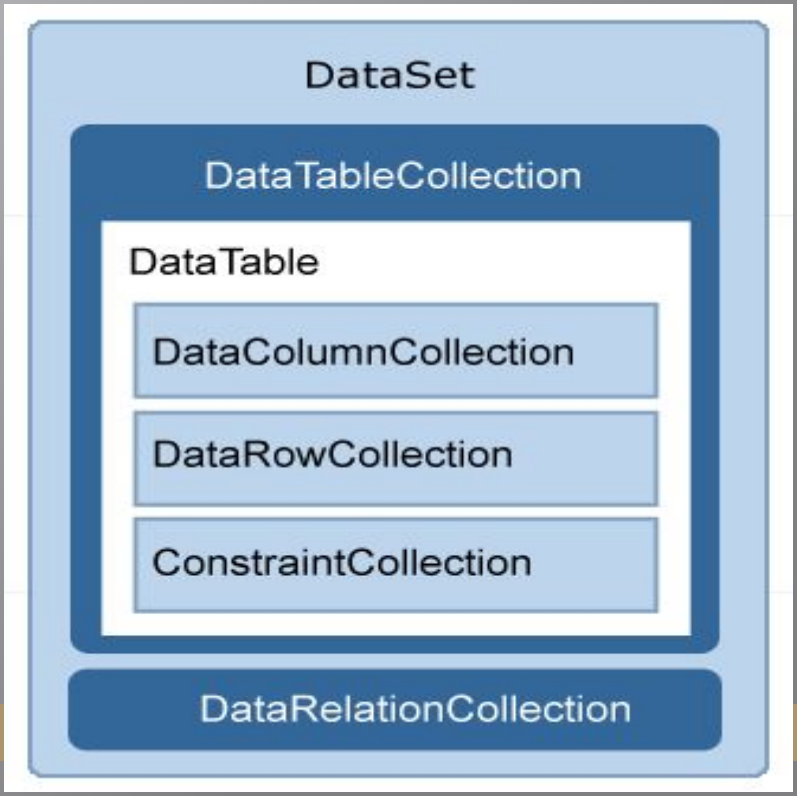
Подключение к базе и считывание данных при каждом запросе может быть дорогостоящей операцией. Зачастую более лучший вариант – считать данные один раз и работать с ними в памяти, а затем сохранить изменения в источнике. Для решения этой задачи и были созданы **отсоединенные наборы данных**:



DataSet – средство локального хранения

DataSet является находящимся в оперативной памяти представлением данных, обеспечивающим согласованную реляционную программную модель *независимо от источника данных*.

DataSet представляет полный набор данных, включая таблицы, содержащие, упорядочивающие и ограничивающие данные, а также связи между таблицами.



Некоторые методы DataSet

Метод	Описание
GetXml() и GetXmlSchema()	Возвращают строку данных (в разметке XML) или информацию схемы для DataSet. Информация схемы — это структурированная информация вроде количества таблиц, их имен, столбцов, типов данных и установленных отношений
WriteXml() и WriteXmlSchema()	Сохраняют данные и схемы, представленные DataSet а файле или потоке формата XML
ReadXml() и ReadXmlSchema()	Создают таблицы в DataSet на основе существующего документа XML или документа схемы XML. Источником XML может быть файл или любой другой поток
Clear()	Очищает все данные таблиц. Однако этот метод оставляет нетронутой информацию о схеме и отношениях
Copy()	Возвращает точный дубликат DataSet с тем же набором таблиц, отношений и данных
Clone()	Возвращает DataSet с той же структурой (таблицами и отношениями), но без данных
Merge()	Принимает другой DataSet, DataTable или коллекцию объектов DataRow и объединяет с текущим объектом DataSet, добавляя новые таблицы и объединяя данные в существующих

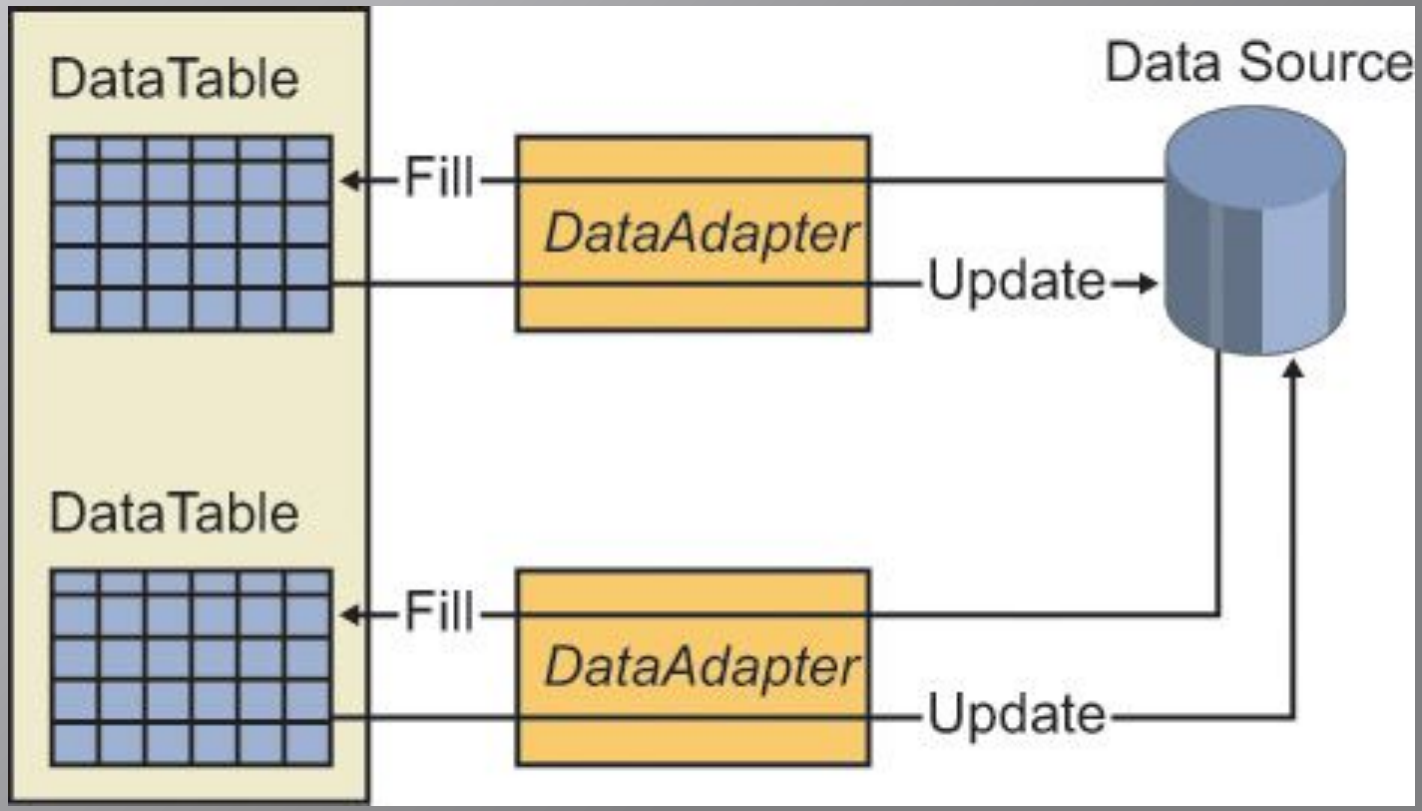
Способы работы с DataSet

Существует несколько способов работы с DataSet, которые могут применяться отдельно или в сочетании. Можно сделать следующее:

1. **Программно**
   создать DataTable, DataRelation и Constraint внутри DataSet и заполнить таблицы данными.
2. **Заполнить** DataSet таблицами данных из ***существующего*** реляционного источника данных с помощью DataAdapter.
3. **Загрузить** и **сохранить** содержимое DataSet с помощью **XML-кода**.

Адаптеры данных

DataAdapter. Класс адаптеров данных применяется для заполнения наборов данных DataSet с помощью объектов DataTable; кроме того, они могут отправлять измененные DataTable назад в базу данных для обработки.



DataAdapter. Свойства и методы.

Методы	
Fill()	Выполняет команду SQL SELECT, указанную в свойстве SelectCommand, для запроса к базе данных и загрузки этих данных в объект DataTable
Update()	Выполняет команды SQL INSERT, UPDATE и DELETE, указанные свойствами InsertCommand, UpdateCommand и DeleteCommand, для сохранения в базе данных изменений, выполненных в DataTable
Свойства	
SelectCommand, InsertCommand, UpdateCommand, DeleteCommand	Содержат SQL-команды, отправляемые в хранилище данных при вызовах методов Fill() и Update()

Наполнение DataSet данными из существующей базы данных

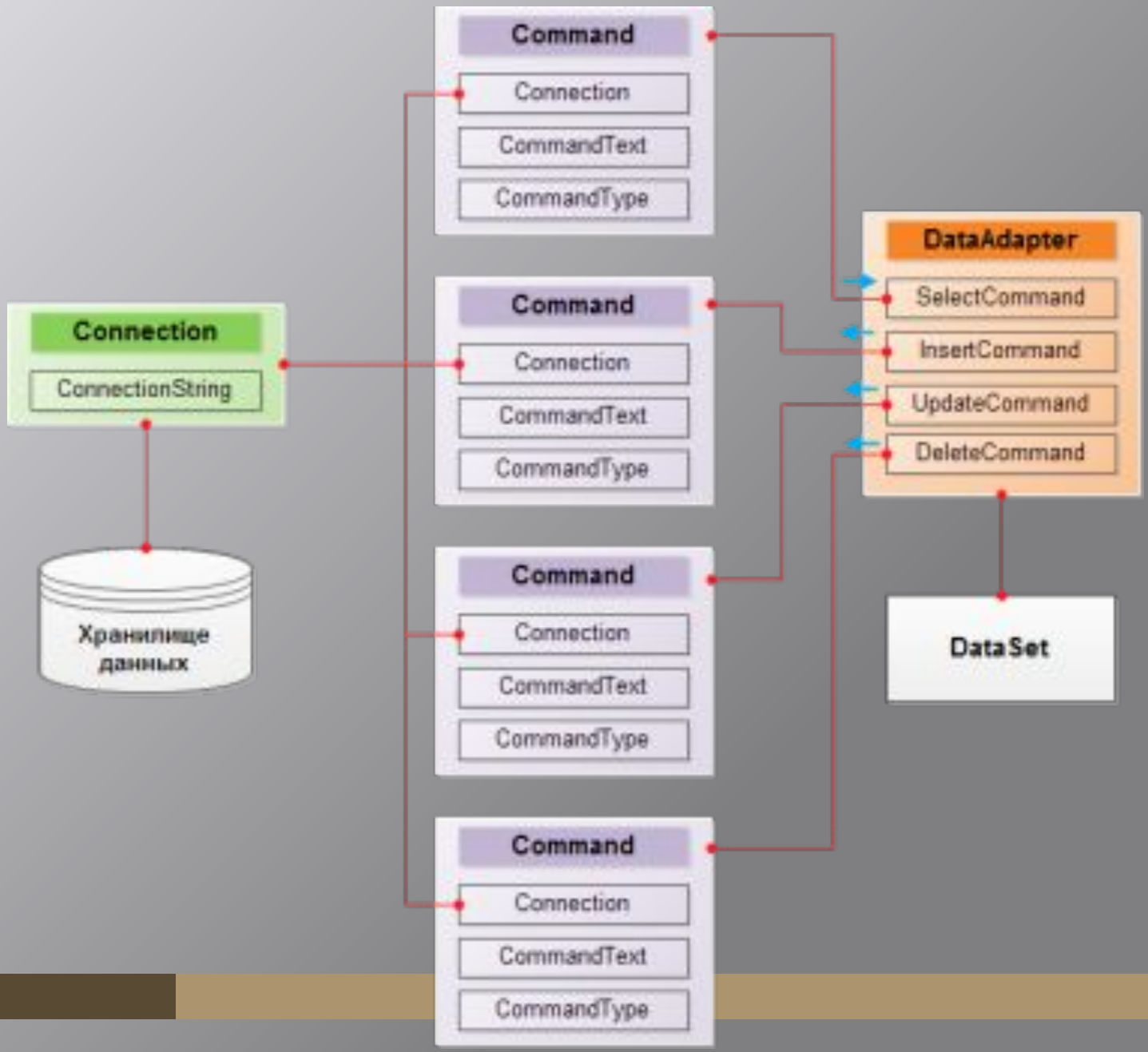
```
SqlConnection conn = new SqlConnection(connectionString);
string command = "SELECT * FROM Employees";

SqlDataAdapter adapter = new SqlDataAdapter(command, conn);

// Заполнить DataSet
DataSet dataset = new DataSet();
adapter.Fill(dataset, "Employees");

// Используем DataSet чтобы вывести все строки таблицы Employees в элемент LabelInfo
foreach (DataRow row in dataset.Tables["Employees"].Rows)
        {
           LabelInfo.Text += String.Format("<li>{0} {1} {2} (<em>{3:d}</em>)<br>",
                    row["EmployeeID"].ToString(), row["LastName"],
                    row["FirstName"], row["BirthDate"]);
        }
```

Манипуляции со строками. DataTables.DataRow

Одна запись в таблице содержится в объекте DataRow, который может хранить **несколько версий записей**:

Значение DataViewRowState	Описание
CurrentRows	Текущие строки, включая не изменившиеся, добавленные и измененные.
Deleted	Удаленная строка.
ModifiedCurrent	Текущая версия, которая является измененной версией исходных данных.
ModifiedOriginal	Исходная версия всех измененных строк. Текущая версия доступна при использовании параметра **ModifiedCurrent**.
Added	Новая строка.
None	Отсутствует.
OriginalRows	Исходные строки, включая неизменившиеся и удаленные.
Unchanged	Неизменившаяся строка.

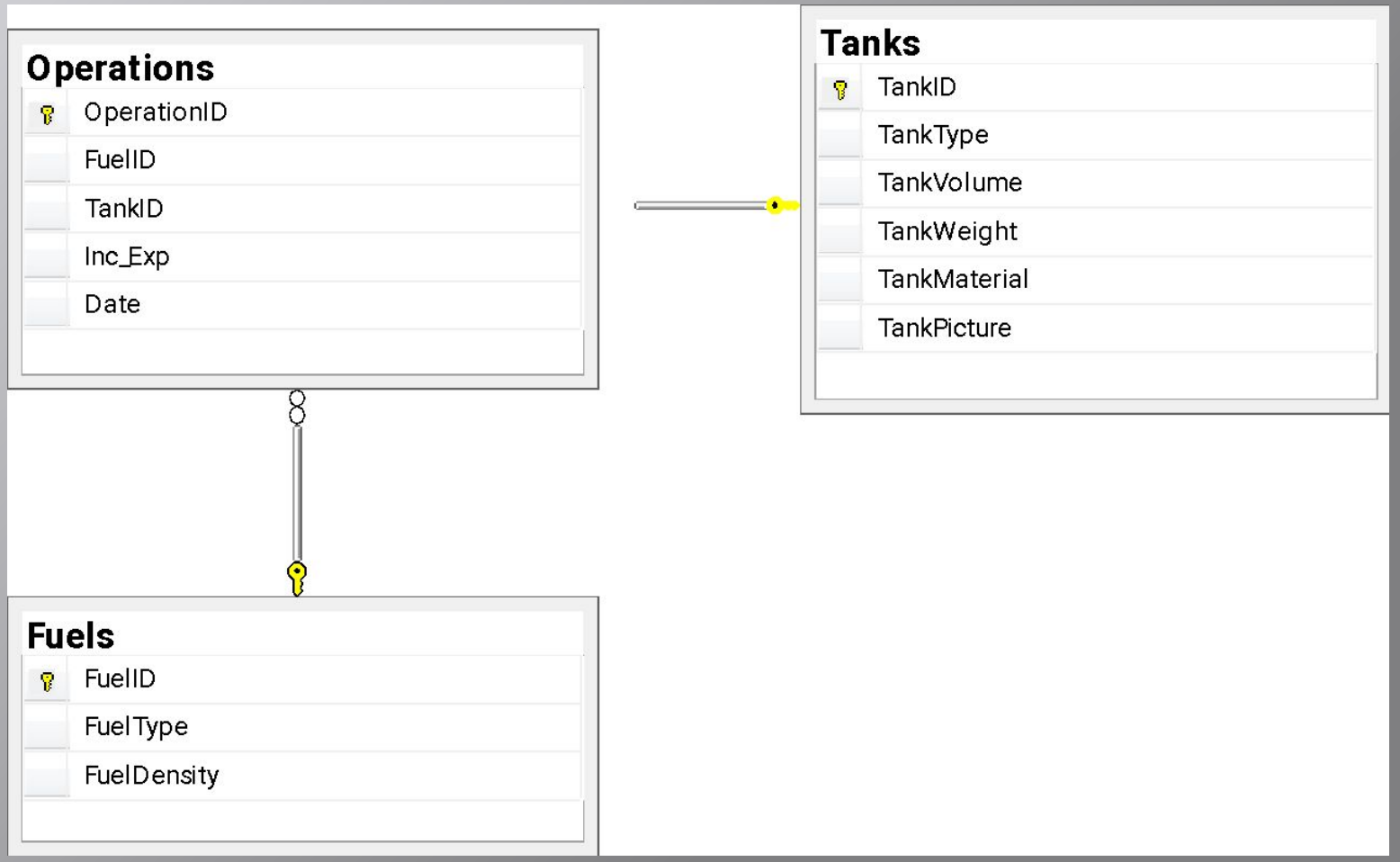
Схема совместной работы объектов ADO .NET



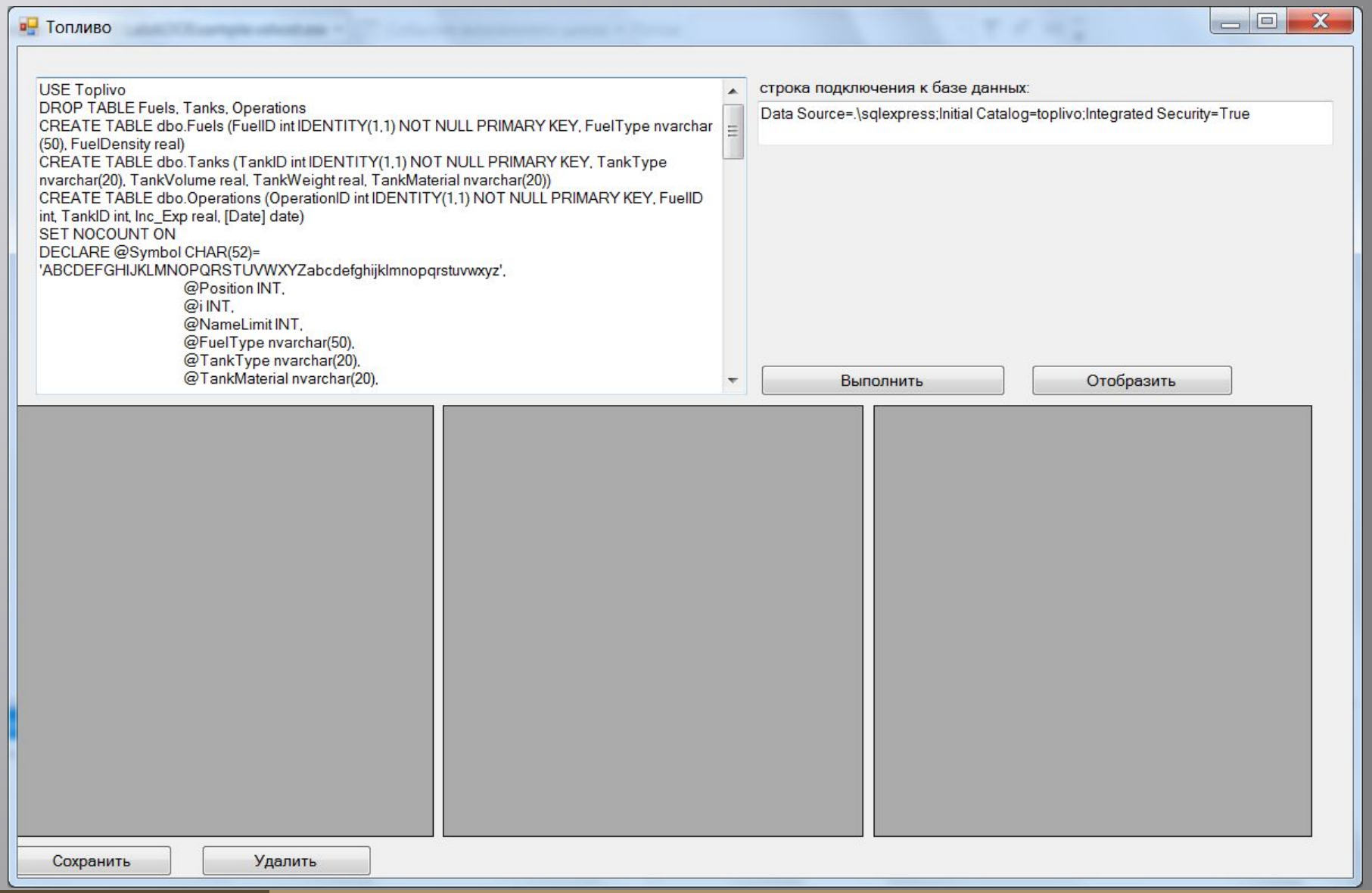
Основные этапы работы с внешними данными через отсоединенный набор ADO.NET

1. **Подключение к существующей базе данных** - создается объект Connection.
2. **Создание DataAdapter** для работы с запрашиваемыми данными.
3. **Создание объект DataSet** для локального хранения данных;
4. **Заполнение данными DataSet** через DataAdapter соединение с источником данных разрывается.
5. **Использование объекта DataSet** клиентской программой для проведения операций: визуализации, изменения, удаления, добавления и т.п. тем или иным способом.
6. **Сохранение изменений во внешнем источнике данных**: объект DataAdapter обращается к объекту Connection и на источник вносятся изменения

Пример. Схема базы данных.



Пример. Инициализация.



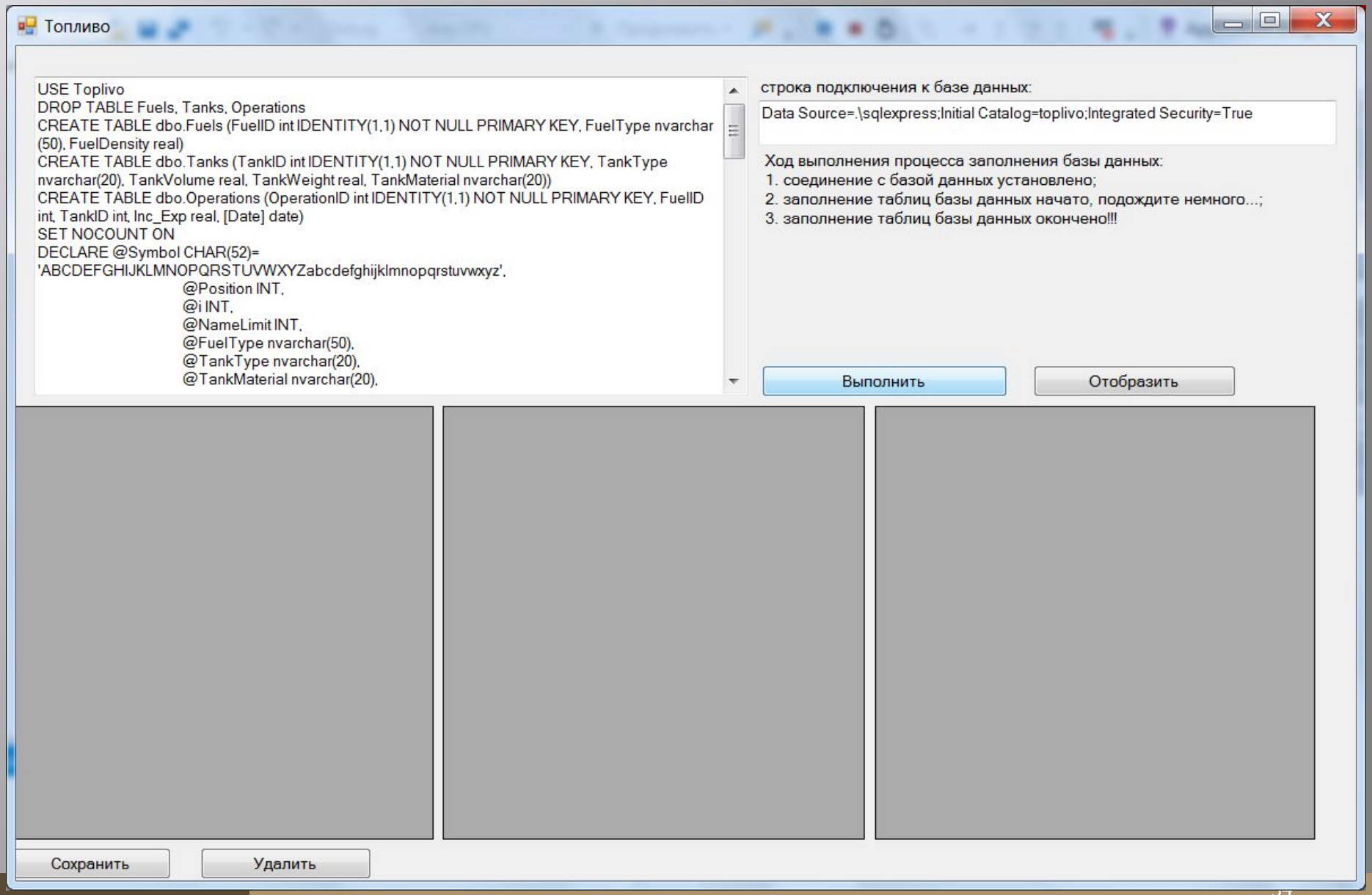
Пример. Инициализация.

Создается рассоединенный набор данных и адаптер.

```
namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {

        SqlDataAdapter dataAdapter = new SqlDataAdapter();
        DataSet ds = new DataSet();


        public Form1()
        {
            InitializeComponent();


        }
```

Исходный код примера: https://github.com/Olgasn/LabADO
https://github.com/Olgasn/LabADO/tree/factory

Пример. Выполнение Command



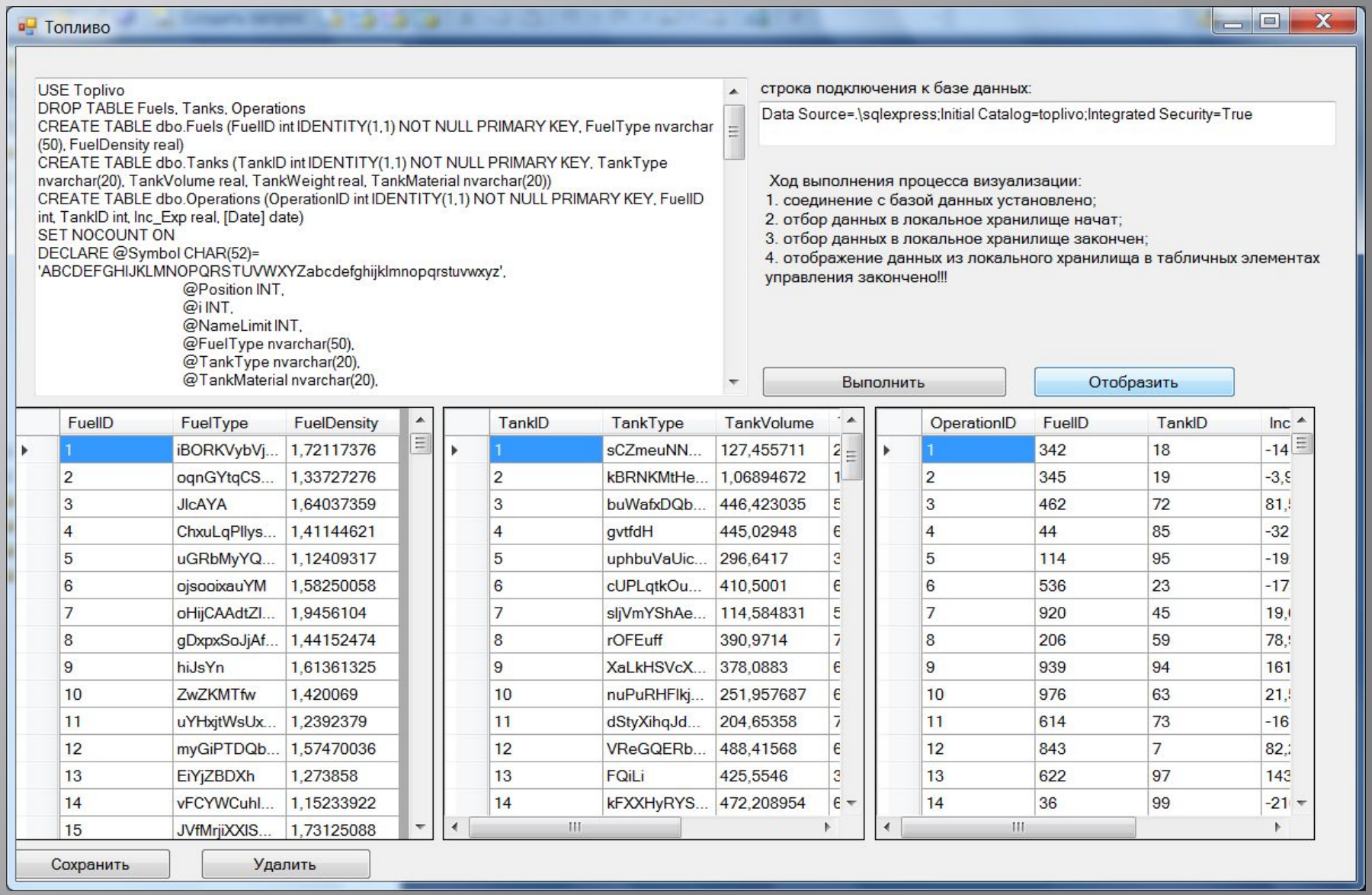
Пример. Выполнение Command

```
private void buttonFill_Click(object sender, EventArgs e)
        try
         {
           string commandText = Convert.ToString(textBoxCommand.Text);
           string ConnectionString = Convert.ToString(textBoxConnectionString.Text);
           SqlConnection conn = new SqlConnection(ConnectionString);
           labelInfo.Text = "Ход выполнения процесса заполнения базы данных:\r\n";
           labelInfo.Refresh();
           try
           {……}
           catch (Exception exeption)
           {
              labelInfo.Text = labelInfo.Text + "Ошибка: "+ exeption.Source;
              labelInfo.Refresh();
           }
          finally
          {
        conn.Close();
        }
}
```

Пример. Выполнение Command.

```
try
        {
            conn.Open();
            labelInfo.Text = labelInfo.Text + "1. соединение с базой данных установлено\r\n";
            labelInfo.Refresh();
            SqlCommand MyCommand = new SqlCommand();
            MyCommand.Connection = conn;
      MyCommand.CommandText = commandText;
            labelInfo.Text = labelInfo.Text + "2. заполнение таблиц базы данных начато, подождите немного...\r\n";
            labelInfo.Refresh();
            MyCommand.ExecuteNonQuery();
            labelInfo.Text = labelInfo.Text + "3. заполнение таблиц базы данных окончено!!!\r\n";
            labelInfo.Refresh();
      }
```

Пример. Наполнение DataSet и визуализация



Пример. Наполнение DataSet

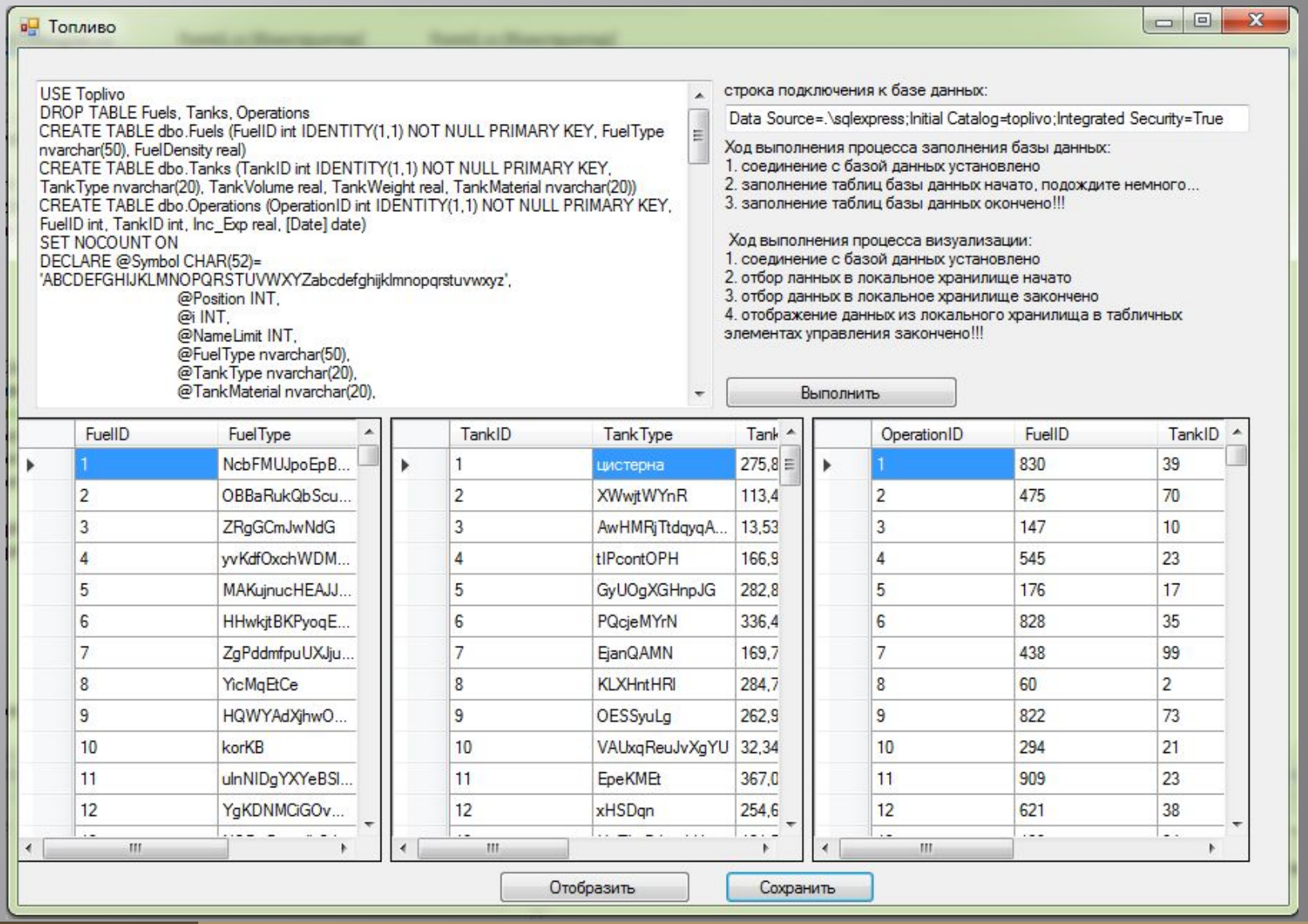
```
conn.Open();
ds.Clear();
labelInfo.Text = labelInfo.Text + "1. соединение с базой данных установлено\r\n";
labelInfo.Refresh();
SqlCommand MyCommand = new SqlCommand();
MyCommand.Connection = conn;
labelInfo.Text = labelInfo.Text + "2. отбор данных в локальное хранилище начат\r\n";
labelInfo.Refresh();
MyCommand.CommandText = "SELECT * FROM Fuels";
dataAdapter.SelectCommand = MyCommand;
if (!(ds.Tables.Contains("Fuels"))) ds.Tables.Add("Fuels"); dataAdapter.Fill(ds, "Fuels");
//
MyCommand.CommandText = "SELECT * FROM Tanks";
dataAdapter.SelectCommand = MyCommand;
if (!(ds.Tables.Contains("Tanks"))) ds.Tables.Add("Tanks"); dataAdapter.Fill(ds, "Tanks");
//
MyCommand.CommandText = "SELECT * FROM Operations";
dataAdapter.SelectCommand = MyCommand;
if (!(ds.Tables.Contains("Operations"))) ds.Tables.Add("Operations");
dataAdapter.Fill(ds, "Operations");
labelInfo.Text = labelInfo.Text + "3. отбор данных в локальное хранилище закончен\r\n";
labelInfo.Refresh();
```

Пример. Визуализация с использованием табличного элемента DataGrid

```
try
        {
            ……
            dataGridView1.DataSource = ds.Tables["Fuels"].DefaultView;
            dataGridView2.DataSource = ds.Tables["Tanks"].DefaultView;
            dataGridView3.DataSource = ds.Tables["Operations"].DefaultView;

            labelInfo.Text = labelInfo.Text + "4. отображение данных из локального хранилища
в табличных элементах управления закончено!!!\r\n";
            labelInfo.Refresh();


}
```

Исходный код примера: https://github.com/Olgasn/LabADO
https://github.com/Olgasn/LabADO/tree/factory

Пример. Изменение и сохранение.



Пример. Сохранение изменённых данных.

```
try
 {
    conn.Open();

    DataTable table = ds.Tables["Fuels"];
    SqlCommand command = new SqlCommand("UPDATE Fuels SET FuelType =
@FuelType, FuelDensity=@FuelDensity WHERE FuelID = @FuelID", conn);
    // Добавление параметров для UpdateCommand.
    command.Parameters.Add("@FuelID", SqlDbType.Int, 5, "FuelID");
    command.Parameters.Add("@FuelType", SqlDbType.NVarChar, 50, "FuelType");
    command.Parameters.Add("@FuelDensity", SqlDbType.Real, 8,
"FuelDensity");

    dataAdapter.UpdateCommand = command;
    dataAdapter.Update(table.Select(null,
null,DataViewRowState.ModifiedCurrent));

  }
```

Исходный код примера: https://github.com/Olgasn/LabADO
https://github.com/Olgasn/LabADO/tree/factory

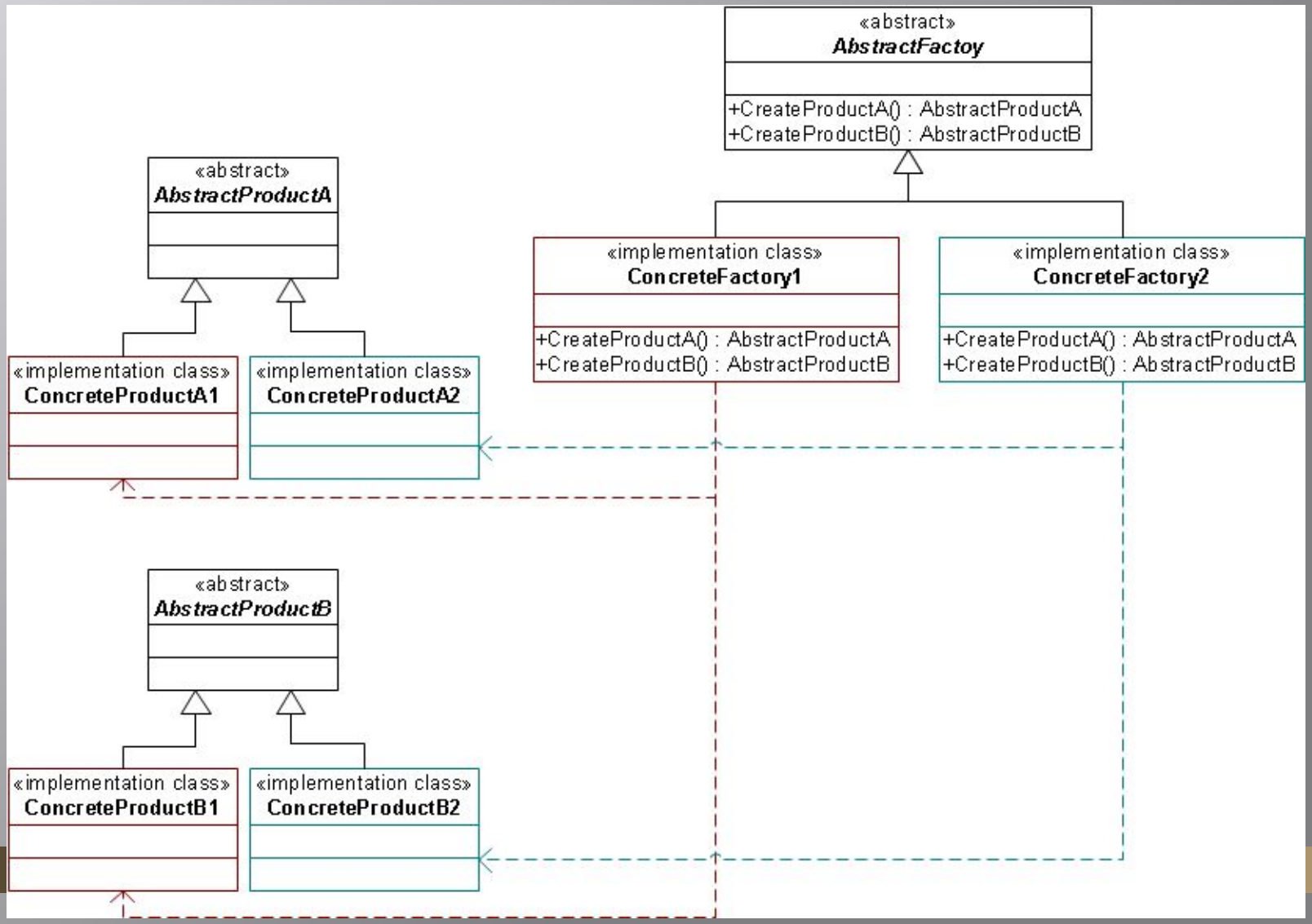
Пример. Удаление данных с использованием DataAdapter.

```
//значение ключевого поля строки для удаления
        int id = (int)dataGridView1.CurrentRow.Cells[0].Value;

        try
        {
            using (conn)
            {
                // Определение строки запроса
                string queryString = "SELECT * FROM Fuels";

                // Создать команду на выборку
                SqlCommand command = new SqlCommand();
                command.CommandText = queryString;
                command.Connection = conn;

                // Создать DbDataAdapter.
                SqlDataAdapter adapter = new SqlDataAdapter();
                adapter.SelectCommand = command;

                // Создать DbCommandBuilder.
                SqlCommandBuilder builder = new SqlCommandBuilder();
                builder.DataAdapter = adapter;
                // Получить команду на удаление
                adapter.DeleteCommand = builder.GetDeleteCommand();

                DataTable table = ds.Tables["Fuels"];

                // Удале
                DataRo
```

Исходный код примера: https://github.com/Olgasn/LabADO
https://github.com/Olgasn/LabADO/tree/factory

Архитектура ADO.NET. Паттерн «Абстрактная фабрика».

В основе модели программирования для написания не зависящего от поставщиков кода лежит использование «**фабричного**» конструктивного шаблона,

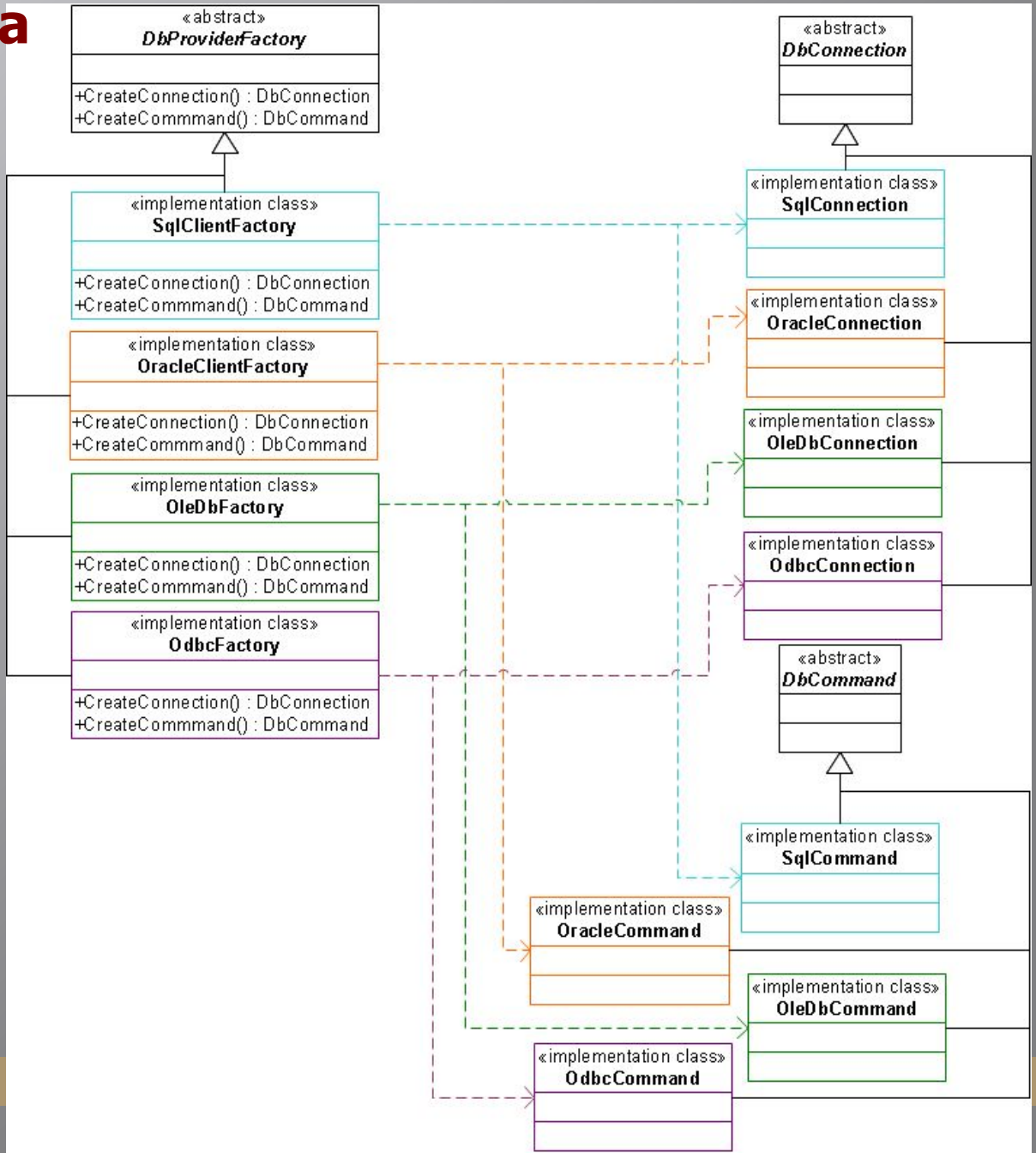


Паттерн «Абстрактная фабрика».

```
abstract class AbstractFactory
{
    public abstract AbstractProductA CreateProductA();
    public abstract AbstractProductB CreateProductB();
}
class ConcreteFactory1 : AbstractFactory
{
    public override AbstractProductA CreateProductA()
    {
        return new ProductA1();
    }

    public override AbstractProductB CreateProductB()
    {
        return new ProductB1();
    }
}
class ConcreteFactory2 : AbstractFactory
{
    public override AbstractProductA CreateProductA()
    {
        return new ProductA2();
    }

    public override AbstractProductB CreateProductB()
    {
        return new ProductB2();
    }
}

abstract class AbstractProductA
{ }

abstract class AbstractProductB
{ }
```

Архитектура ADO.NET. Реализация паттерна абстрактная фабрика

В «**фабричном**» **конструктивном шаблоне** используется один API-интерфейс для доступа к базам данных нескольких поставщиков. Этому шаблону присваивается соответствующее имя, поскольку он задает использование специализированного объекта, только чтобы создавать другие объекты, что очень напоминает настоящую фабрику.



Создание фабрики (factory). Класс DbProviderFactories

Фабрика классов сама по себе специфична для поставщика. Например, поставщик SQL Server включает класс System.Data.SqlClient.SqlClientFactory.

Поставщик Oracle использует System.Data.OracleClient.OracleClientFactory.

Однако существует полностью стандартизованный класс, который предназначен для **динамического нахождения и создания необходимой фабрики**.

Класс **System.Data.Common.DbProviderFactories** - предоставляет статический метод **GetFactory**(), возвращающий фабрику на основе имени поставщика.

Например, вот как выглядит код, использующий **DbProviderFactories** для получения SqlClientFactory:

```
// классы DbProviderFactories и DbProviderFactory
// находятся в пространстве имён System.Data.Common
var f=DbProviderFactories.GetFactory("System.Data.SqlClient");
DbCommand cmd = f.CreateCommand();
```

Создание фабрики (factory). Класс DbProviderFactories

DbProviderFactories – класс представляет набор статических методов для создания одного или нескольких экземпляров классов DbProviderFactory.
Методы

Имя	Описание
GetFactory(DataRow)	Возвращает экземпляр класса DbProviderFactory.
GetFactory(DbConnection)	Возвращает экземпляр класса DbProviderFactory.
GetFactory(String)	Возвращает экземпляр класса DbProviderFactory.
GetFactoryClasses()	Возвращает объект DataTable, содержащий сведения обо всех установленных поставщиках, реализующих объект DbProviderFactory.

Создание фабрики. Класс DbProviderFactories

DbProviderFactory – класс содержит набор методов для создания экземпляров классов поставщиков, реализующих источник данных.

Некоторые методы

Имя	Описание
CreateCommand()	Возвращает новый экземпляра класса поставщика, реализующий класс DbCommand.
CreateCommandBuilder()	Возвращает новый экземпляра класса поставщика, реализующий класс DbCommandBuilder.
CreateConnection()	Возвращает новый экземпляра класса поставщика, реализующий класс DbConnection.
CreateConnectionStringBuilder()	Возвращает новый экземпляра класса поставщика, реализующий класс DbConnectionStringBuilder.
CreateDataAdapter()	Возвращает новый экземпляра класса поставщика, реализующий класс DbDataAdapter.
CreateDataSourceEnumerator()	Возвращает новый экземпляра класса поставщика, реализующий класс DbDataSourceEnumerator.
CreateParameter()	Возвращает новый экземпляра класса поставщика, реализующий класс DbParameter.

Использование фабрики (factory). Пример

```
DbProviderFactory  factory = DbProviderFactories.GetFactory("System.Data.SqlClient");
connection = factory.CreateConnection();
connection.ConnectionString = connectionString;

if (connection != null)

        {
            using (connection)
            {
                try
                {
                    // Открыть соединение.
                    connection.Open();
                    // Создать и выполнить DbCommand.
                    DbCommand command = connection.CreateCommand();
                    command.CommandText ="INSERT INTO Categories (CategoryName)
VALUES ('Low Carb')";
                    int rows = command.ExecuteNonQuery();
        }
                    // Обработка ошибок, связанных с данными.
                    catch (DbException exDb)
                    {}
                    // Обработка других ошибок.
                    catch (Exception ex)
                    {}
```

Использование фабрики (factory). Пример обработки ошибок.

```
// Обработка ошибок, связанных с данными.
catch (DbException exDb)
            {
      Console.WriteLine("DbException.GetType: {0}", exDb.GetType());
      Console.WriteLine("DbException.Source: {0}", exDb.Source);
            Console.WriteLine("DbException.ErrorCode: {0}", exDb.ErrorCode);
            Console.WriteLine("DbException.Message: {0}", exDb.Message);
             }
// Обработка других ошибок.
catch (Exception ex)
            {
             Console.WriteLine("Сообщение об исключении: {0}", ex.Message);
             }
```

Реализация типовых операций

- **Создание DbProviderFactory и DbConnection**

Создание фабрики DbProviderFactory и соединения DbConnection путем передачи имени поставщика в формате «System.Data.ProviderName» и строки соединения.

- **Выборка данных с использованием объекта DbCommand**

Объект DbCommand создается для выбора данных из таблицы Categories путем задания CommandText инструкции SQL SELECT. Предполагается, что в источнике данных существует таблица Categories. Открывается соединение, и данные получаются при помощи объекта DbDataReader.

- **Выполнения команды с использованием DbCommand**
- **Выборка данных с помощью объекта DbDataAdapter**
- **Изменение данных с помощью DbDataAdapter**

Модификация данных в DataTable с использованием DbDataAdapter, в котором применяется объект DbCommandBuilder для формирования команд, необходимых для обновления данных в источнике данных.

Коды

https://github.com/Olgasn/UsingDbProviderFactory

СПАСИБО ЗА ВНИМАНИЕ!

ВОПРОСЫ?

Технологии доступа к данным. ADO .NET

Author: Олег Асенчик