Основные технологии Microsoft ADO.NET

(практика)

Санкт-Петербург

2017

В данном учебно-методическом пособии приведены материалы практических занятий по курсу «Основные технологии Microsoft ADO.NET».

Целью практических занятий является приобретение навыков по применению технологии ADO.NET.

Составитель: Осипов Никита Алексеевич, ктн, доцент, MCT, MCITP, MCTS.

СОДЕРЖАНИЕ

Работа в подключенном режиме

Практическое занятие 1. Подключение к базе данных

За подключение и непосредственную работу с базами данных отвечают поставщики данных. В .NET используются два поставщика данных: SQL Client .NET Data Provider и OLE DB .NET Data Provider.

Первый из них предназначен для работы только с базами данных Microsoft SQL Server. За счет узкой направленности только на одну базу данных от одного производителя классы и код провайдера могут быть оптимизированы для максимально эффективной работы с сервером.

Компания Microsoft реализовала универсальный провайдер OLE DB .NET Data Provider, который позволяет подключиться к любой базе данных, для которой есть поставщик данных OLE DB.

Классы для работы с SQL Client.NET Data Provider находятся в пространстве имен `System.Data.SqlClient`, а классы OLE DB.NET Data Provider расположены в `System.Data.OleDb`.

Для удобства разработки классы обоих провайдеров реализованы схожим образом и наследуются от одного и того же базового класса. Это значит, что методы работы с данными будут идентичными.

Упражнение 1. Создание строки подключения

Для соединения с базой данных в упражнениях этого занятия используется класс `OleDbConnection`. Этому классу нужно указать с помощью строки подключения (`Connection String`) параметры подключения к базе данных. Из этой строки компонент узнает, где находится база данных, и какие параметры нужно использовать для подключения и авторизации.

Строку подключения можно написать самостоятельно вручную, а можно использовать встроенное в ADO окно создания строки.

1. Чтобы воспользоваться удобным окном создания строки подключения, создайте в любом месте на диске файл с расширением `udl`. Это можно сделать в Проводнике или в любом другом файловом менеджере. Имя файла и его расположение не имеют никакого значения.
2. Запустите созданный файл, и появится диалоговое окно *Свойства связи с данными* для редактирования строки подключения.

3. Для использования базы данных SQL на вкладке *Поставщик данных* выберите драйвер Microsoft OLE DB Provider for SQL Server.
4. На вкладке *Подключение* вверху в пункте 1 есть выпадающий список, в котором выберите SQL-сервер, доступный в вашей сети. Если в списке не найден нужный вам сервер, то его можно ввести непосредственно в поле ввода.
5. В пункте 2 определяются параметры авторизации, т. е. учетная запись, которая будет использоваться для подключения. Используйте параметры текущей записи, под которой вы вошли в систему (можно также указать имя и пароль явно).
6. В пункте 3 выберите имя базы данных на сервере - `Northwind`. С помощью кнопки *Проверить подключение* проверьте установленное подключение. В случае успешной проверки нажмите OK.
7. Верните расширение файла обратно на txt, откройте его и просмотрите строку подключения.

Строка подключения к базе данных `Northwind` например, может выглядеть следующим образом:

```
Provider=SQLOLEDB.1;Integrated Security=SSPI;Persist Security
Info=False;Initial Catalog=Northwind;Data Source=(local)
```

В этой строке подключения указываются следующие параметры:

`Provider` — провайдер, через который будет происходить подключение;

`Integrated Security` — если в строке этот параметр равен SSPI, то при авторизации будет использоваться текущее имя пользователя и пароль, под которым вы авторизовались в системе;

`Persist Security Info` — сохранять пароль в строке подключения. Это удобно, потому что не нужно каждый раз вводить пароль, но абсолютно небезопасно;

`Initial Catalog` — имя базы данных;

`Data Source` — имя сервера базы данных.

Упражнение 2. Создание соединения с базой данных

В этом упражнении для соединения с базой данных используется класс `OleDbConnection`, для которого нужно указать с помощью строки подключения (`Connection String`) параметры подключения к базе данных.

1. Создайте новое WinForms-приложение (назовите его DBConnection) и добавьте на форму меню, в котором сделайте пункт для подключения к базе данных.

2. Объявите объект класса `OleDbConnection` в качестве члена класса формы следующим образом:

```
OleDbConnection connection = new OleDbConnection();
```

3. Объявите переменную, хранящую строку подключения к базе данных, которая будет использоваться для поиска сервера базы данных:

```
string testConnect = @"Provider=SQLOLEDB.1;Integrated
Security=SSPI;Persist Security Info=False;Initial Catalog=Northwind;Data
Source=(local)";
```

4. В обработчике события `click` пункта меню укажите для свойства `ConnectionString` переменную, хранящую строку подключения к базе данных, и вызовите метод `open()`.

В итоге получится следующий код:

```
try
     {
       if (connection.State != ConnectionState.Open)
         {
            connection.ConnectionString = testConnect;
            connection.Open();
MessageBox.Show("Соединение с базой данных выполнено успешно");
          }
        else
MessageBox.Show("Соединение с базой данных уже установлено");
      }
     catch
           {
MessageBox.Show("Ошибка соединения с базой данных");
           }
```

Метод `open()` не возвращает никаких значений, но может сгенерировать исключения:

`InvalidoperationException` – соединение уже открыто;

`OleDbException` – ошибка уровня соединения с базой данных.

Обрабатывать ошибки необходимо, потому что при соединении с базой данных ошибки могут возникать достаточно часто. Могут быть проблемы с сетью, сервер может зависнуть и находиться в процессе перезагрузки, могут возникнуть проблемы с авторизацией и т. д. Возможных проблем при работе с базами данных очень много, поэтому оставлять вызов метода подключения без обработки исключительных ситуаций не стоит.

5. Хорошим стилем является явное закрытие соединения с базой данных. Создайте пункт меню *Отключить соединение*, который будет соединяться с сервером и закрывать соединение.
6. Добавьте код обработчика для закрытия соединения:

```
if (connection.State == ConnectionState.Open)
  {
     connection.Close();
     MessageBox.Show("Соединение с базой данных закрыто");
  }
else
     MessageBox.Show("Соединение с базой данных уже закрыто");
```

7. Запустите и протестируйте приложение.

Упражнение 3. Обработка ошибок

В практических задачах желательно, чтобы разрабатываемые приложения были более удобными в использовании и предоставляли подробное описание возникающих ошибок. В этом случае служба поддержки сможет быстрее справиться с проблемами.

В этом упражнении Вы добавите код в приложение предыдущего упражнения для обработки исключений соединения и будете показывать описание и уровень (severity level) для каждого типа ошибки `SqlError`, перехваченного исключением.

1. Импортируйте пространство имен:

```
using System.Data.SqlClient;
```

2. Добавьте после блока `try` обработчик блок `catch` для перехвата `OleDbException`. Пройдитесь в цикле по коллекции `Errors` и для каждого элемента создавайте окно сообщения, в котором отображается подробное описании ошибки и ее код:

```
catch (OleDbException XcpSQL)
            {
                foreach (OleDbError se in XcpSQL.Errors)
                {
                    MessageBox.Show(se.Message,
                    "SQL Error code " + se.NativeError,
                    MessageBoxButtons.OK,
                    MessageBoxIcon.Information);
                }
            }
```

3. Вместо универсального обработчика `catch` без параметров добавьте обработчик код для перехвата всех других типов исключений и отображения для них общего сообщения:

```
catch (Exception Xcp)
    {
       MessageBox.Show(Xcp.Message, "Unexpected Exception",
                       MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
```

4. Внесите ошибки в строку подключения, например, измените имя провайдера, имя базы данных (ошибка 4060), имя сервера (ошибка 17) и любые другие ошибки на ваше усмотрение. Протестируйте работу программы. Изучите содержание диалоговых окон, информирующих об ошибках подключения.

Упражнение 4. Событие `StateChange`

При каждой смене состояния подключения (с открытого на закрытое, и обратно) объекты подключения становятся источником события `StateChange`. В этом упражнении Вы добавите в код предыдущего упражнения, который обрабатывает событие `StateChange` объекта `OleDbConnection`.

1. Добавьте обработчик события `StateChange` объекта `connection` в конструктор формы:

```
this.connection.StateChange += new
                System.Data.StateChangeEventHandler(
                this.connection_StateChange);
```

2. Создайте обработчик события `StateChange` объекта `connection` таким образом, чтобы он проверял текущее состояние соединения и делал доступными/недоступными соответствующие пункты меню:

```
  private void connection_StateChange(object sender,
System.Data.StateChangeEventArgs e)
         {
             соединитьсяСБазойToolStripMenuItem.Enabled =
                 (e.CurrentState == ConnectionState.Closed);
             оключитьСоединениеToolStripMenuItem.Enabled =
                 (e.CurrentState == ConnectionState.Open);
         }
```

3. Протестируйте приложение. Проверьте, что в определенном состоянии подключения соответствующие пункты меню становятся недоступными.

Упражнение 5. Работа с классами конфигурации

Строки соединения можно сохранить в файлах конфигурации, что исключает необходимость внедрять их в код приложения. Файлы конфигурации представляют собой стандартные XML-файлы, для которых в платформе .NET Framework определен типовой набор элементов.

В пространстве имен System.Configuration есть классы, упрощающие извлечение строк подключения из файлов конфигурации во время выполнения. Предусмотрена возможность получить строку соединения программным путем по ее имени или по имени поставщика.

Для работы с файлами конфигурации на локальном компьютере используется класс **ConfigurationManager.**

Получение строк соединения во время выполнения

Выполняется итерация в коллекции ConnectionStringSettings и отображение свойств `Name`, `ProviderName` и `ConnectionString` в окне сообщения:

`Name` – имя строки соединения, сопоставляется с атрибутом **name**.

`ProviderName` – полное имя поставщика, сопоставляется с атрибутом **providerName**.

`ConnectionString` – строка соединения, сопоставляется с атрибутом **connectionString**.

1. Откройте приложение из прошлого упражнения.
2. Добавьте ссылку на библиотеку System.Configuration.dll и подключите требуемое пространство имен:

```
using System.Configuration;
```

3. Добавьте новый пункт *Список Подключений* в меню Файл.
4. Добавьте обработчик события `click` нового пункта меню и реализуйте в нем следующую функциональность:
    a. Создайте экземпляр коллекции ConnectionStringSettings и укажите ему свойство `ConnectionStrings` класса **ConfigurationManager:**

```
ConnectionStringSettingsCollection settings =
ConfigurationManager.ConnectionStrings;
```

    b. В цикле просмотрите коллекцию ConnectionStringSettings и выведете значения свойств `Name`, `ProviderName` и `ConnectionString` в окна сообщения:

```
if (settings != null)
{
```

```
            foreach (ConnectionStringSettings cs in settings)
            {
        MessageBox.Show("name = " + cs.Name);
        MessageBox.Show("providerName = " + cs.ProviderName);
        MessageBox.Show("connectionString = " + cs.ConnectionString);
            }
        }
```

5. Запустите приложение. Выберете команду *Список Подключений* и изучите содержимое окон сообщений.

Извлечение строки соединения по имени

Ниже рассматривается способ получения строки соединения из файла конфигурации путем указания ее имени. Будет создан объект `ConnectionStringSettings`, сопоставляя указанный входной параметр с именем `ConnectionStrings`. Если совпадающее имя не найдено, метод возвратит значение `null`.

6. Добавьте в проект файл конфигурации приложения (меню **Проект → Добавить новый элемент**), оставьте имя по умолчанию – App.config.
7. Откройте файл конфигурации и между тегами `<configuration>` добавьте информацию о подключении (значение для свойства `connectionString` скопируйте у переменной `testConnect`):

```
<connectionStrings>
  <add name="DBConnect.NorthwindConnectionString"
      connectionString="Provider=SQLOLEDB.1;Integrated
Security=SSPI;Persist Security Info=False;Initial Catalog=Northwind;Data
Source=(local)"
      providerName="System.Data.OleDb" />
</connectionStrings>
```

8. В коде формы закомментируйте инициализацию переменной соединения `testConnect`.
9. После конструктора формы добавьте статический метод, возвращающий значение строки соединения по ее имени:

```
static string GetConnectionStringByName(string name)
{
    string returnValue = null;
    ConnectionStringSettings settings =
        ConfigurationManager.ConnectionStrings[name];
    if (settings != null)
        returnValue = settings.ConnectionString;
```

```
            return returnValue;
        }
```

10. После метода объявите переменную `testConnect` и присвойте ей значение, возвращаемое из метода путем передачи ему в качестве аргумента значение свойства `name` (см. файл конфигурации приложения):

```
  string testConnect =
GetConnectionStringByName("DBConnect.NorthwindConnectionString");
```

11. Запустите приложение и попробуйте выполнить соединение с базой данных. Теперь значение строки подключения берется из файла конфигурации. Должно появиться окно, подтверждающее успешность соединения.

Практическое занятие 2. Выполнение команд

В предыдущих упражнениях было показан процесс подключения к серверу. После подключения к базе данных обычно требуется получить определенные данные. Это можно реализовать с помощью выполняемых на сервере команд. Для выполнения команд используются объекты класса `OleDbCommand`. У конструктора нет параметров, достаточно проинициализировать объект значением по умолчанию. После этого в свойство `commandText` нужно поместить SQL-запрос, и выполнить его.

Для выполнения запросов существует несколько методов. Все зависит от того, какой результат требуется получить.

Упражнение 1. Получение скалярного значения

В этом упражнении Вы реализуете возможность получения одиночного значения.

1. Откройте приложение, построенное в предыдущем упражнении.
2. Добавьте на форму кнопку `Button` и текстовый элемент `Label`. Для обоих элементов свойству `Text` укажите значение *Сколько продуктов*.
3. В обработчике события `Click` сначала проверьте наличие соединения и создайте объект `OleDbCommand`.
4. Установите в свойстве `Connection` требуемый объект соединения.
5. В свойстве укажите `CommandText` текст запроса на подсчет числа записей в таблице `Products`.

6. Объявите переменную целого типа и для выполнения SQL-команды используйте метод `ExecuteScalar()`. Он подходит для тех случаев, когда запрос возвращает только одно значение. Результат метода `ExecuteScalar()` универсален и имеет тип данных `object`, и в данном случае его следует явно привести к числовому типу.
7. Результат запроса верните в свойство `Text` элемента `Label`.

В итоге код может иметь следующий вид:

```
if (connection.State == ConnectionState.Closed)
            {
                MessageBox.Show("Сначала подключитесь к базе");
                return;
            }
            OleDbCommand command = new OleDbCommand();
            command.Connection = connection;
            command.CommandText = "SELECT COUNT(*) FROM Products";
            int number = (int)command.ExecuteScalar();
            label1.Text = number.ToString();
```

Протестируйте приложение.

Упражнение 2. Получение набора данных

Для выполнения запросов, возвращающих наборы данных, используется класс `OleDbCommand`. Метод `ExecuteReader()` выполняет запрос и возвращает объект класса `OleDbDataReader`, через который можно просмотреть набор данных результата.

В этом упражнении реализуется возможность считывания требуемого содержимого таблицы `Products` в компонент `Listview`.

1. Откройте приложение, построенное в предыдущем упражнении.
2. Добавьте на форму кнопку `Button` и компонент `Listview`.
3. Для кнопки свойству `Text` укажите значение *Список продуктов*.
4. Для `Listview`:
    a. свойству `Size` установите размер `180;170`,
    b. в свойстве `Columns` добавьте столбец с заголовком (свойство `text`): *Название продукта*,
    c. свойству `View` – вид отображения `Details`.
5. В обработчике события `Click` желательно сначала проверить наличие соединения (как в прошлом упражнении) и создайте объект `OleDbCommand`. В отличие от прошлого упражнения для создания объекта `oleDBCommand` используйте метод `CreateCommand()` объекта соединения.

Этот метод инициализирует новый объект для выполнения команд, в качестве соединения устанавливает себя и возвращает созданный объект в качестве результата:

```
OleDbCommand command = connection.CreateCommand();
```

В прошлом упражнении Вы проинициализировали переменную самостоятельно конструктором класса `OleDbCommand` и установили в свойство `connection` нужный объект соединения. Оба способа идентичны.

6. В свойстве укажите `CommandText` текст запроса на выборку продуктов (поле `ProductName`) в таблице `Products`.
7. Для того чтобы получить набор данных выполните SQL-команду с помощью метода `ExecuteReader()`. Она возвращает в качестве результата объект `OleDbDataReader`, через который и читаются данные результата.

```
OleDbDataReader reader = command.ExecuteReader();
```

8. Для считывания данных построчно используйте цикл, а для получения очередной строки данных вызовите `Read()` класса `OleDbDataReader`. Этот метод возвращает булево значение, которое определяет, прочиталась ли очередная строка. Если мы достигли конца набора данных, то результатом вызова метода будет `false`:

```
while (reader.Read())
     {
        listView1.Items.Add(reader["ProductName"].ToString());
     }
```

9. В итоге код может иметь следующий вид:

```
OleDbCommand command = connection.CreateCommand();
 command.CommandText = "SELECT ProductName FROM Products";
 OleDbDataReader reader = command.ExecuteReader();
   while (reader.Read())
     {
       listView1.Items.Add(reader["ProductName"].ToString());
     }
```

10. Запустите и протестируйте приложение.

Упражнение 3. Выполнение транзакций

Транзакция состоит из одной команды или группы команд, которые выполняются как пакет. Транзакции позволяют объединить несколько операций в одну единицу работы. Если в какой-либо точке транзакции

возникает ошибка, может быть выполнен откат всех обновлений к их состоянию до начала транзакции. Для выполнения локальной (состоит из одной фазы и обрабатывается непосредственно базой данных) транзакции каждый поставщик данных платформы .NET Framework имеет свой собственный объект `Transaction`.

В этом упражнении инициируется транзакция с помощью метода `BeginTransaction` объекта `Connection`. После начала транзакции при помощи свойства `Transaction` объекта `Command` к ней прикрепляется команда и затем в зависимости от успеха или ошибки компонентов транзакция фиксируется или откатывается.

1. Откройте приложение, построенное в предыдущем упражнении.
2. Добавьте на форму кнопку `Button`. Для свойства `Text` укажите значение *Транзакция*.
3. Реализуйте в обработчике события `Click` новой кнопки следующую функциональность:
    a. Создайте объект соединения – объект класса `OleDbConnection`, назначьте строку подключения и откройте соединение:

```
OleDbConnection connection = new OleDbConnection(testConnect);
      connection.Open();
```

    b. Вызовите метод `BeginTransaction()` объекта соединения `OleDbConnection` для отметки начала транзакции. Данный метод возвращает ссылку на транзакцию. Эта ссылка назначается объектам `OleDbCommand`, прикрепленным к транзакции:

```
OleDbTransaction OleTran = connection.BeginTransaction();
```

    c. Создайте объект `OleDbCommand`, с помощью которого будет выполняться SQL-команда:

```
OleDbCommand command = connection.CreateCommand();
```

    d. Свяжите транзакцию с командой. Для этого объект транзакции поместите в свойство `Transaction` объекта команды:

```
command.Transaction = OleTran;
```

    e. В `try`-блоке реализуйте следующее: свойству `commandText` укажите запрос, который нужно выполнить (команда INSERT, которая вставляет в таблицу `Products` название нового продукта). Для выполнения команд, не возвращающих результатов, а изменяющих данные или вставляющих данные в таблицы, используйте метод `ExecuteNonQuery()`. Для сохранения изменений, сделанных внутри транзакции,

вызовите метод `Commit()` объекта `OleDbTransaction`. Итоговый код для вставки двух строк будет следующим:

```
    try
    {
    command.CommandText =
  "INSERT INTO Products (ProductName) VALUES('Wrong size')";
        command.ExecuteNonQuery();
    command.CommandText =
   "INSERT INTO Products (ProductName) VALUES('Wrong color')";
        command.ExecuteNonQuery();

        OleTran.Commit();
MessageBox.Show("Both records were written to database");
    }
```

f. В `catch`-обработчике вызовите окно с сообщением об ошибке в случае ее возникновения:

```
    catch (Exception ex)
    {
       MessageBox.Show(ex.Message);
        ...
    }
```

g. В этом же блоке `catch` вызовите метод `Rollback()` для отмены изменений, т. е. отката транзакции. Метод вызовите во вложенном блоке `try`:

```
        try
        {
            OleTran.Rollback();
        }
        catch (Exception exRollback)
        {
            MessageBox.Show(exRollback.Message);
        }
```

h. Закройте соединение:

```
  connection.Close();
```

4. Запустите и протестируйте приложение. По клику кнопки *Транзакция* в таблицу продуктов добавляются новые строки. Проверьте это с помощью кнопки *Список продуктов*.

5. Внесите ошибку в код вставки строк в таблицу, например, неправильное название столбца или таблицы. Повторите запись строк в таблицу и убедитесь в откате транзакции.

Практическое занятие 3. Подключение к данным с помощью IDE

На практике при разработке приложений, предназначенных для работы с базой данных удобно использовать возможности интегрированной среды разработки и компоненты панели **Toolbox.**

Упражнение 1. Создание подключения к БД в Server Explorer

В этом упражнении описывается процесс создания узла Data Connection в Server Explorer.

1. Создайте новое WinForms-приложение (назовите его DSConnect).
2. С помощью меню **View** откройте окно **Server Explorer**.
3. Щелкните правой кнопкой мыши узел **Data Connections** и выберите команду **Add Connection**. При первом подключении появится диалоговое окно **Change Data Source.**
4. В окне **Data source** выберите источник данных – Microsoft SQL Server. В соответствии с выбранным источником данных, выпадающий список **Data provider** заполняется подходящими провайдерами данных. Выберите .NET Framework Data Provider for SQL Server.
5. В окне **Add Connection** в поле Server name выберите свой сервер или введите имя вручную.
6. Выберите метод аутентификации для доступа к серверу – Windows.
7. Выберите требуемую базу данных – `Northwind`. Проверьте подключение и в случае удачного исхода нажмите OK.
8. После создания подключения просмотрите его свойства в окне **Properties**.

Упражнение 2. Создание подключений с помощью мастера

В этом упражнении используется способ создания подключений с помощью мастера Data Source Configuration Wizard.

1. Выберите в меню **Data** команду **Add New Data Source**.
2. В окне **Choose a Data Source Type** оставьте тип источника данных по умолчанию **Database**.
3. В окне **Choose a Database Model** выберите **Dataset**.

4. В окне **Choose a Your Data Connection** проверьте, что созданное ранее подключение отображается в списке подключений. Для создания нового подключения нажмите кнопку **New Connection**…
5. С помощью окна **Add Connection** создайте подключение по аналогии с добавлением подключения в прошлом упражнении.
6. В окне **Save the Connection String to …** сохраните строку подключения в конфигурационном файле приложения и укажите имя подключения.
7. Страница **Choose Your Database Objects** позволяет выбирать таблицы, представления, хранимые процедуры для работы в приложении. Раскройте узел **Tables** и выберите таблицы **Customers** и **Orders**.
8. Щелкните **Finish**, таким образом, типизированный набор данных с указанным в мастере добавлен в проект.

Упражнение 3. Добавление объектов ADO.NET в WinForms приложение

Создайте новое WinForms-приложение (назовите его DBCommand).

В следующих упражнениях используются поставщик данных SQL Client.NET Data Provider. Добавьте на панель инструментов требуемые для дальнейшей работы компоненты .Net Framework: `SqlCommand`, `SqlCommandBuilder`, `SqlConnection`, `SqlDataAdapter`.

Добавление в приложение объекта SqlConnection

В этом разделе, вы добавите объект `SqlConnection` в ваше приложение и настроите строку соединения на подключение к БД Northwind на локальном SQL сервере.

1. Перетащите объект `SqlConnection` на рабочую область формы. В результате будет создан объект `SqlConnection` с именем `SqlConnection1`.
2. В окне **Properties** объекта найдите свойство `ConnectionString`**.** Раскройте выпадающий список напротив имени свойства и выберите **<New Connection.>**.
3. В появившемся окне укажите свой сервер, например, в предыдущих упражнениях – (local) (у вас может быть другое имя, будьте внимательны) и выберите требуемую базу данных – `Northwind`. Проверьте подключение и в случае удачного исхода нажмите OK.
4. Проверьте, что свойство `ConnectionString` объекта `SqlConnection1` имеет значение строки подключения.

Добавление объекта SqlCommand в приложение

В этом разделе вы добавите в приложение объект `SqlCommand` и настроите его так, чтобы он мог исполнять SQL запрос через соединение `SqlConnection1`.

1. Перейдите на вкладку **Data** панели **Toolbox**, и перетащите объект `SqlCommand` на форму. В результате в приложении, будет создан объект `SqlCommand1`.
2. В окне **Properties** объекта `SqlCommand1` укажите для свойства `Connection` значение `SqlConnection1`.
3. Свойство `CommandText`, содержащий запрос на выборку данных настроим в следующем упражнении.

Упражнение 4. Исполнение запроса и отображение результата

В этом упражнении реализуете несколько команд на выполнение запросов в БД Northwind с помощью объектов **SqlConnection** и **SqlCommand**.

Выполнение простого запроса

Реализуйте функциональность для выполнения команды, которая заполняет текстовое поле списком значений `CustomerID` и `CompanyName` из таблицы `Customers`.

1. Импортируйте пространство имен:

```
using System.Data.SqlClient;
```

2. Добавьте на форму кнопку. Установите свойству **Text** значение *Запрос данных*.
3. В окне **Properties** укажите для свойства `CommandText` объекта **SqlCommand1** значение, содержащее запрос на выборку данных:

```
SELECT CustomerID, CompanyName FROM Customers
```

4. Реализуйте в обработчике события `Click` следующую функциональность:
    a. Создайте объект `StringBuilder` для хранения результата запроса:

```
System.Text.StringBuilder results = new System.Text.StringBuilder();
```

    b. Укажите, что требуется выполнить инструкцию SQL:

```
sqlCommand1.CommandType = CommandType.Text;
```

    c. Непосредственно перед выполнением команды откройте подключение:

```
sqlConnection1.Open();
```

    d. Присвойте результаты инструкции объекту `SqlDataReader`:

```
SqlDataReader reader = sqlCommand1.ExecuteReader();
bool MoreResults = false;
do
     {
         while (reader.Read())
         {
             for (int i = 0; i < reader.FieldCount; i++)
             {
                 results.Append(reader[i].ToString() + "\t");
             }
             results.Append(Environment.NewLine);
         }
         MoreResults = reader.NextResult();
     } while (MoreResults);
MoreResults = reader.NextResult();
} while (MoreResults);
```

e. Закройте объект `SqlDataReader` и подключение:

```
reader.Close();
sqlCommand1.Connection.Close();
```

f. Выведете результаты в текстовое поле:

```
ResultsTextBox.Text = results.ToString();
```

5. Запустите и протестируйте приложение.

Вызов хранимой процедуры

Реализуйте функциональность для выполнения команды, которая запускает хранимую процедуру «`Ten Most Expensive Products`».

1. Добавьте на форму новую кнопку. Установите свойству `Text` значение *Вызов процедуры*.
2. Добавьте на форму новый объект `SqlCommand` на форму. В результате в приложении, будет создан объект `SqlCommand2`.
3. В окне **Properties** объекта `SqlCommand2` укажите для свойства `Connection` значение `SqlConnection1`.
4. Реализуйте в обработчике события `Click` требуемую функциональность по аналогии с предыдущим обработчиком, учитывая, что различие между вызовом команды SQL и вызовом хранимой процедуры состоит в том, что для вызова процедуры значение свойства `CommandType` устанавливается равным `StoredProcedure`, а в свойстве `CommandText` указывается имя хранимой процедуры.

5. В итоге обработчик события `Click` будет выглядеть следующим образом:

```
System.Text.StringBuilder results = new System.Text.StringBuilder();
     sqlCommand2.CommandType = CommandType.StoredProcedure;
     sqlCommand2.CommandText = "Ten Most Expensive Products";
     sqlCommand2.Connection.Open();
     SqlDataReader reader = sqlCommand2.ExecuteReader();
         while (reader.Read())
         {
             for (int i = 0; i < reader.FieldCount; i++)
             {
                 results.Append(reader[i].ToString() + "\t");
             }
             results.Append(Environment.NewLine);
         }
         reader.Close();
         sqlCommand2.Connection.Close();
         ResultsTextBox.Text = results.ToString();
```

6. Запустите и протестируйте приложение. Обратите внимание, что в первом случае свойство `CommandText` инициализировалось с помощью окна **Properties,** а во втором непосредственно в коде. Определите, какой способ является предпочтительным.

Выполнение команды, выполняющую операцию с каталогом

Добавьте новую функциональность – возможность выполнить команду для создания новой таблицы в базе данных. Для выполнения такой операции необходимо применить метод `ExecuteNonQuery()` объекта `Command`.

1. Добавьте на форму новую кнопку. Установите свойству `Text` значение *Создание таблицы*.
2. Добавьте на форму новый объект `SqlCommand` на форму. В результате в приложении, будет создан объект `SqlCommand3`.
3. В окне **Properties** объекта `SqlCommand3` укажите для свойства `Connection` значение `SqlConnection1`.
4. Реализуйте в обработчике события `Click` требуемую функциональность, учитывая, что для выполнения команды, запускающей выражения SQL вызывается метод ExecuteNonQuery() и нет необходимости использовать SqlDataReader, поскольку команда не возвращает никаких данных.

5. В итоге обработчик события `Click` будет выглядеть следующим образом:

```
sqlCommand3.CommandType = CommandType.Text;
sqlCommand3.CommandText = "CREATE TABLE SalesPersons (" +
       "[SalesPersonID] [int] IDENTITY(1,1) NOT NULL, " +
       "[FirstName] [nvarchar](50)  NULL, " +
       "[LastName] [nvarchar](50)  NULL)";
sqlCommand3.Connection.Open();
sqlCommand3.ExecuteNonQuery();
sqlCommand3.Connection.Close();
MessageBox.Show("Таблица SalesPersons создана");
```

6. Запустите и протестируйте приложение. В окне Server Explorer проверьте, что таблица успешно добавлена.
7. Попробуйте повторно добавить таблицу. Должно сгенерироваться исключение. Нажмите *Продолжить* и попробуйте выполнить предыдущую хранимую процедуру. Должно снова сгенерироваться исключение, так как соединение с базой данных не было закрыто.
8. Реализуйте обработчик исключительной ситуации с возможностью определения типа исключения. Перенесите в блок `finally` код для закрытия соединения.

Выполнение запроса с параметром

Добавьте в приложение возможность пользователю выполнить параметризованный запрос на выборку данных.

1. Добавьте на форму **TextBox** и установите свойству **Name** значение `CityTextBox`, а свойству **Text** – `London`.
2. Добавьте на форму новую кнопку и установите свойству **Text** значение *Запрос с параметром*.
3. Добавьте на форму новый объект `SqlCommand` на форму. В результате в приложении, будет создан объект `SqlCommand4`.
4. В окне **Properties** объекта `SqlCommand4` укажите для свойства `Connection` значение `SqlConnection1`.
5. В окне **Properties** укажите для свойства `CommandText` объекта `SqlCommand4` значение, содержащее запрос на выборку данных с параметром:

```
SELECT CustomerID, CompanyName, City FROM Customers WHERE City = @City
```

6. В окне **Properties** для свойства `Parameters` откройте окно добавления параметров. Проверьте, что в окне редактора коллекции автоматически

добавлен параметр (если не добавлен, то добавьте самостоятельно) `@City`, для `SqlDbType` установите значение `NVarChar`.

7. Реализуйте в обработчике события `Click` требуемую функциональность по аналогии с простым запросом и перед открытием соединения задайте параметру значение, введенное в текстовое поле:

```
sqlCommand4.Parameters["@City"].Value = CityTextBox.Text;
```

8. В итоге обработчик события `Click` будет выглядеть следующим образом:

```
System.Text.StringBuilder results = new System.Text.StringBuilder();
            sqlCommand4.CommandType = CommandType.Text;
            sqlCommand4.Parameters["@City"].Value = CityTextBox.Text;
            sqlConnection1.Open();
            SqlDataReader reader = sqlCommand4.ExecuteReader();
            bool MoreResults = false;
            do
            {
                while (reader.Read())
                {
                    for (int i = 0; i < reader.FieldCount; i++)
                    {
                        results.Append(reader[i].ToString() + "\t");
                    }
                    results.Append(Environment.NewLine);
                }
                MoreResults = reader.NextResult();
            } while (MoreResults);
            reader.Close();
            sqlCommand4.Connection.Close();
            ResultsTextBox.Text = results.ToString();
```

9. Запустите и протестируйте приложение. Введите в поле запроса Madrid и повторно выполните запрос.

Выполнение параметризованной хранимой процедуры

Реализуйте функциональность для выполнения команды, которая запускает параметризованную хранимую процедуру «`SalesByCategory`».

1. Добавьте на форму новую кнопку и установите свойству **Text** значение *Процедура с параметром*.

2. Добавьте на форму **TextBox** и установите свойству **Name** значение `CategoryNameTextBox`, а свойству **Text** – `Beverages`.
3. Добавьте еще **TextBox** и установите свойству **Name** значение `OrdYearTextBox`, а свойству **Text** – 1997.
4. Добавьте на форму новый объект `SqlCommand` на форму. В результате в приложении, будет создан объект `SqlCommand5`.
5. В окне **Properties** объекта `SqlCommand5` укажите:
    a. для свойства `Connection` значение `SqlConnection1`.
    b. для свойства `CommandType` значение `StoredProcedure`.
    c. для свойства `CommandText` имя хранимой процедуры: `SalesByCategory`.
6. На вопрос о создании коллекции **Parameters** ответьте *Да*.
7. В окне **Properties** для свойства `Parameters` откройте окно добавления параметров. Проверьте, что в окне редактора коллекции автоматически добавлены параметры `@CategoryName` и `@OrdYear`.
8. Реализуйте в обработчике события `Click` требуемую функциональность по аналогии с простой процедурой и перед открытием соединения задайте параметрам значения, вводимые в текстовые поля:

```
sqlCommand5.Parameters["@CategoryName"].Value = CategoryNameTextBox.Text;
sqlCommand5.Parameters["@OrdYear"].Value = OrdYearTextBox.Text;
```

9. В итоге обработчик события `Click` будет выглядеть следующим образом:

```
System.Text.StringBuilder results = new System.Text.StringBuilder();
sqlCommand5.Parameters["@CategoryName"].Value = CategoryNameTextBox.Text;
sqlCommand5.Parameters["@OrdYear"].Value = OrdYearTextBox.Text;
            sqlCommand5.Connection.Open();
            SqlDataReader reader = sqlCommand5.ExecuteReader();
            while (reader.Read())
            {
                for (int i = 0; i < reader.FieldCount; i++)
                {
                    results.Append(reader[i].ToString() + "\t");
                }
                results.Append(Environment.NewLine);
            }
            reader.Close();
            sqlCommand5.Connection.Close();
```

```
ResultsTextBox.Text = results.ToString();
```

10. Запустите приложение и протестируйте его. Введите имя другой категории, например, Condiments, Seafood, Produce или другой год, например, 1998.

Работа в отключенной среде

В данном модуле рассматривается главный элемент автономной архитектуры ADO.NET – класс **DataSet**. Этот класс позволяет загрузить из источника часть данных и работает как контейнер данных, обеспечивающий реляционную структуру автономной природы и механизм хранения данных, который не зависит от источника данных. Помимо обеспечения интуитивного и легкого доступа к данным, **DataSet** также хранит историю изменений вплоть до изменений отдельных ячеек, реализовывая, таким образом, параллельную работу и сохранение изменений обратно в источник данных.

Практическое занятие 4. Создание объектов DataSet

Объекты **DataSet** доступны в пространстве имен System.Data и используются в приложении для кеширования данных в памяти. **DataSet** содержит объекты **DataTable**, которые могут быть связаны с помощью объектов **DataRelation** подобно структуре реляционной базы данных.

Упражнение 1. Использование DataSet Designer для создания DataSet

В этом упражнении применяется средство времени разработки, с помощью которого упрощается процесс создания типизированных объектов DataSet.

1. Создайте новое WinForms-приложение (назовите его DatasetDesigner).
2. В меню **Project** выберите команду **Add New Item** и добавьте шаблон **DataSet** (Набор данных), укажите ему имя NorthwindDataSet.xsd.
3. Перенесите таблицы **Customers** и **Orders** из окна **Server Explorer** на поверхность конструктора.
4. Постройте проект.
5. На поверхность формы проекта перенесите в левый верхний угол формы кнопку и установите свойству (**Name**) значение `GetCustomersButton`, свойству **Text** – *Get Customers*.
6. На поверхность формы проекта перенесите ЭУ **ListBox**, свойству (**Name**) укажите значение `CustomersListBox`. Сделайте так, чтобы компонент занимал нижнюю половину формы.

7. Реализуйте в обработчике события `Click` следующую функциональность:
    a. Создайте экземпляр типизированного набора данных Northwind:

```
NorthwindDataSet NorthwindDataset1 = new NorthwindDataSet();
```

    b. Создайте экземпляр `CustomersTableAdapter`

```
NorthwindDataSetTableAdapters.CustomersTableAdapter
CustomersTableAdapter1 =
                new NorthwindDataSetTableAdapters.CustomersTableAdapter();
```

    c. Вызовите метод для загрузки всех клиентов в DataTable:

```
CustomersTableAdapter1.Fill(NorthwindDataset1.Customers);
```

    d. Передайте значения столбца `CompanyName` в **ListBox**:

```
foreach (NorthwindDataSet.CustomersRow NWCustomer in
                            NorthwindDataset1.Customers.Rows)
        {
            CustomersListBox.Items.Add(NWCustomer.CompanyName);
        }
```

8. Выполните приложение. Проверьте, что значение `CompanyName` каждого клиента отображается в списке.

Упражнение 2. Создание объектов DataTable

В этом упражнении создаются и настраиваются объекты **DataTable**, которые обеспечивают хранение в памяти данных приложения подобно таблице базы данных. Процессы заполнения **DataTable** и управления данными рассматриваются позже.

1. Создайте новое WinForms-приложение (назовите его CreatingDataTable).
2. На поверхность формы проекта перенесите в левый верхний угол формы кнопку и установите свойству (**Name**) значение `AddRowButton`, свойству **Text** – *Add Row*.
3. На поверхность формы проекта перенесите ЭУ **DataGridView**, свойству (**Name**) укажите значение `TableGrid`. Сделайте так, чтобы компонент занимал нижнюю половину формы.
4. В поле класса формы создайте экземпляр таблицы:

```
private DataTable CustomersTable = new DataTable("Customers");
```

5. Создать таблицу можно несколькими способами: указать код инициализации непосредственно в конструкторе формы, создать для этого обработчик события загрузки формы (`Load`) или в обработчике

события `Click` специальной кнопки. Выберем второй вариант. Создайте обработчик события `Load` формы.

6. В обработчике загрузки формы:
    a. Присвойте **DataGridView** отображаемой таблице:

```
TableGrid.DataSource = CustomersTable;
```

    b. Определите схему таблицы добавлением столбцов (объектов **DataColumn**) к коллекции `Columns` таблицы:

```
CustomersTable.Columns.Add("CustomerID", Type.GetType("System.String"));
CustomersTable.Columns.Add("CompanyName", Type.GetType("System.String"));
CustomersTable.Columns.Add("ContactName", Type.GetType("System.String"));
CustomersTable.Columns.Add("ContactTitle", Type.GetType("System.String"));
CustomersTable.Columns.Add("Address", Type.GetType("System.String"));
CustomersTable.Columns.Add("City", Type.GetType("System.String"));
CustomersTable.Columns.Add("Country", Type.GetType("System.String"));
CustomersTable.Columns.Add("Phone", Type.GetType("System.String"));
```

    c. Укажите столбец `CustomerID` как первичный ключ:

```
DataColumn[] KeyColumns = new DataColumn[1];
KeyColumns[0] = CustomersTable.Columns["CustomerID"];
CustomersTable.PrimaryKey = KeyColumns;
```

    d. Запретите для столбцов `CustomerID` и `CompanyName` значения `Null`:

```
CustomersTable.Columns["CustomerID"].AllowDBNull = false;
CustomersTable.Columns["CompanyName"].AllowDBNull = false;
```

7. В обработчике события `Click` специальной кнопки *Add Row:*
    a. Реализуйте создание записи:

```
DataRow CustRow = CustomersTable.NewRow();
Object[] CustRecord =  {"ALFKI", "Alfreds Futterkiste", "Maria Anders",
                "Sales Representative", "Obere Str. 57", "Berlin",
                  null, "12209", "Germany", "030-0074321","030-0076545"};
CustRow.ItemArray = CustRecord;
```

    b. Добавьте запись в таблицу:

```
CustomersTable.Rows.Add(CustRow);
```

8. Запустите приложение. После загрузки формы отобразится созданная таблица. Добавьте строку в таблицу. Проверьте, что повторная попытка добавить строку приводит к исключительной ситуации. Реализуйте обработчик исключения с возможностью определения типа исключения.
9. Исключительная ситуация возникает по причине настройки столбца `CustomerID` как первичного ключа. Закомментируйте соответствующий

код и проверьте, что теперь можно повторно добавлять одинаковые строки.

Упражнение 3. Создание объектов DataAdapter

Объекты **DataAdapter** содержат помимо сведений о подключении, команды, необходимые для выборки и обновления данных в базе данных, а также для заполнения объектов **DataSet** и **DataTable**. Объекты **DataAdapter** можно создавать визуально, используя Data Adapter Configuration Wizard (Мастер настройки адаптера данных) или программно, создав экземпляр требуемого адаптера провайдера и передав конструктору предложение SELECT и допустимый объект `Connection`.

В этом упражнении показывается процесс создания объектов **DataAdapter** и работа с ними.

1. Создайте новое WinForms-приложение (назовите его DataAdapterWizard).
2. Добавьте (если ранее не был добавлен) на панель инструментов требуемый для дальнейшей работы компонент `SqlDataAdapter.`
3. Перенесите на форму объект `SqlDataAdapter,` откроется Data Adapter Configuration Wizard (Мастер настройки адаптера данных).
4. На странице **Choose Your Data Connection** (Выбор подключения базы данных) выберите подключение к БД Northwind.
5. На странице **Choose a Command Type** (Выбор типа команды) оставьте настройку по умолчанию (использовать инструкции SQL).
6. На странице **Generate the SQL statement** (Создание инструкций SQL) введите следующую инструкцию:

```
SELECT * FROM Customers
```

7. Нажмите **Next** (Далее) и изучите результаты работы мастера. Нажмите **Finish** (Готово) для завершения работы мастера и добавления `SqlDataAdapter` к форме.
8. Сгенерируйте типизированный **DataSet**, основанный на настроенном адаптере, для чего в меню **Data** (Данные) выберите команду **Generate DataSet** (Создать набор данных). Обратите внимание, что выбран настроенный адаптер.
9. Для нового набора данных введите имя **NorthwindDataSet** и нажмите OK.
10. Перенесите **DataGridView** на форму. Сделайте так, чтобы компонент занимал верхнюю половину формы.

11. Создайте обработчик события `Load` формы.
12. В обработчике загрузки формы:
    a. Настройте **DataGridView** для отображения таблицы клиентов:

```
dataGridView1.DataSource = northwindDataSet1.Customers;
```

    b. Вызовите метод `Fill` объекта **DataAdapter** для загрузки клиентов данными:

```
sqlDataAdapter1.Fill(northwindDataSet1.Customers);
```

13. Запустите приложение проверьте, что требуемая таблица появляется в сетке.
10. Добавьте на форму кнопку и установите свойству (**Name**) значение `UpdateButton`, свойству **Text** – *Save Change.*
14. В обработчике события `Click` вызовите метод адаптера данных `Update` для сохранения изменений в базе данных:

```
sqlDataAdapter1.Update(northwindDataSet1);
```

15. Запустите приложение. Добавьте новую информацию в таблицу. В поле `CustomerID` введите WERTY, в остальные поля по своему усмотрению. Сохраните изменения и закройте приложение. Запустите приложение и проверьте наличие добавленной строки.

Обработка событий DataAdapter

Объекты **DataAdapter** вызывают несколько событий, которые могут использоваться при выполнении операций по отношению к источнику данных:

`RowUpdating` – создается после каждого **DataRow** во время обновления;

`RowUpdated` – создается перед каждым **DataRow** во время обновления;

`FillError` – создается при возникновении ошибки во время выполнения Fill.

1. Создайте обработчик события `RowUpdating` и реализуйте следующую функциональность:
    a. Создайте экземпляр `CustomerRow` и присвойте ему изменяемую строку:

```
  NorthwindDataSet.CustomersRow CustRow =
(NorthwindDataSet.CustomersRow)e.Row;
```

    b. Отобразите диалоговое окно для подтверждения обновления:

```
  DialogResult response = MessageBox.Show("Continue updating " +
CustRow.CustomerID.ToString() +
                  "?", "Continue Update?", MessageBoxButtons.YesNo);
```

    c. Реализуйте отмену обновления, если пользователь выбрал Нет:

```
  if (response == DialogResult.No)
```

```
            {
                e.Status = UpdateStatus.SkipCurrentRow;
            }
```

2. Создайте обработчик события `RowUpdated` и реализуйте следующую функциональность:
    a. Создайте экземпляр `CustomerRow` и присвойте ему изменяемую строку:

```
  NorthwindDataSet.CustomersRow CustRow =
(NorthwindDataSet.CustomersRow)e.Row;
  MessageBox.Show(CustRow.CustomerID.ToString() + " has been updated");
```

    b. Реализуйте сброс таблицы после обновления строки для отражения изменений:

```
  northwindDataSet1.Customers.Clear();
  sqlDataAdapter1.Fill(northwindDataSet1.Customers);
```

3. Создайте обработчик события `FillError` и реализуйте следующую функциональность:
    a. Отобразите диалоговое окно для реагирования на ошибку:

```
  DialogResult response = MessageBox.Show("The following error occurred
while Filling the DataSet: "+ e.Errors.Message.ToString() +
   " Continue attempting to fill?", "FillError Encountered",
MessageBoxButtons.YesNo);
```

    b. Реализуйте попытку продолжения, если пользователь выбрал Да:

```
  if (response == DialogResult.Yes)
            {
                e.Continue = true;
            }
            else
            {
                e.Continue = false;
            }
```

4. Запустите приложение протестируйте его, изменяя значения в записях таблицы. Обратите внимание, что можно индивидуально обработать каждую конкретную строку, которую адаптер пытается обновить.

Упражнение 4. Программное создание объекта DataAdapter

1. Создайте новое WinForms-приложение (назовите его DataAdapterProgram).

2. Добавьте на форму **DataGridView**. Сделайте так, чтобы компонент занимал верхнюю половину формы.
3. Добавьте пространство имен `using System.Data.SqlClient`;
4. Программно добавьте в класс формы объект `Connection` для соединения с базой данных на локальном сервере:

```
private SqlConnection NorthwindConnection = new SqlConnection("Data
Source=(local);Initial Catalog=Northwind;Integrated Security=True");
```

5. Добавьте после соединения код для создания **DataAdapter**:

```
private SqlDataAdapter SqlDataAdapter1;
```

6. Создайте объект **DataSet** с таблицей `Customer`:

```
private DataSet NorthwindDataset = new DataSet("Northwind");
private DataTable CustomersTable = new DataTable("Customers");
```

7. Создайте обработчик события `Load` формы в нем укажите следующий код:

```
SqlDataAdapter1 = new SqlDataAdapter("SELECT * FROM Customers",
NorthwindConnection);
NorthwindDataset.Tables.Add(CustomersTable);
SqlDataAdapter1.Fill(NorthwindDataset.Tables["Customers"]);
dataGridView1.DataSource = NorthwindDataset.Tables["Customers"];
```

8. Создайте там же объект `CommandBuilder`, который предоставит дополнительные команды INSERT, UPDATE и DELETE, требующиеся для обновления базы при вызове метода `DataAdapter.Update`:

```
SqlCommandBuilder commands = new SqlCommandBuilder(SqlDataAdapter1);
```

9. Добавьте на форму кнопку и установите свойству (**Name**) значение `UpdateButton`, свойству **Text** – *Save Change.*
10. В обработчике события `Click` вызовите метод адаптера данных `Update` для сохранения изменений в базе данных:

```
NorthwindDataset.EndInit();
SqlDataAdapter1.Update(NorthwindDataset.Tables["Customers"]);
```

11. Запустите приложение измените в сетке значение в одной из записей. Сохраните изменения и проверьте сохранение данных при повторном запуске приложения.

Упражнение 5. Обработка данных в DataTable

В упражнении описывается загрузка данных в таблицу, удаление строк из таблицы и редактирование текущих значений, а также просмотр сведений **RowState** и **DataRowVersion** для записи в **DataTable**.

1. Создайте новое WinForms-приложение (назовите его WorkingDataTable).
2. Добавьте пространство имен `using System.Data.SqlClient;`
3. Добавьте на форму **DataGridView**. Сделайте так, чтобы компонент занимал верхнюю половину формы. Измените его свойство (**Name**) на `CustomersDataGridView`.
4. Добавьте на форму кнопку и установите свойству (**Name**) значение `FillTableButton`, свойству **Text** – *Fill Table.*
5. Перенесите на форму компонент **SqlDataAdapter**, откроется мастер настройки адаптера. На первой странице укажите подключение к БД Northwind.
6. При выборе типа команды оставьте по умолчанию – использовать инструкции SQL.
7. Введите инструкцию `SELECT * FROM Customers`. Проверьте, что в области компонентов появился объект **Connection**.
8. Сгенерируйте типизированный **DataSet**, основанный на настроенном адаптере, для чего в меню **Data** (Данные) выберите команду **Generate DataSet** (Создать набор данных). Обратите внимание, что выбран настроенный адаптер. Для нового набора данных введите имя **NorthwindDataSet**.
9. Создайте обработчик события `Load` формы в нем укажите следующий код:

```
CustomersDataGridView.DataSource = northwindDataSet1.Customers;
CustomersDataGridView.MultiSelect = false;
CustomersDataGridView.SelectionMode =
DataGridViewSelectionMode.CellSelect;
CustomersDataGridView.EditMode =
DataGridViewEditMode.EditProgrammatically;
```

10. В обработчике события `Click` кнопки **Fill Table** вызовите метод адаптера данных `Fill`:

```
sqlDataAdapter1.Fill(northwindDataSet1.Customers);
```

11. Добавьте на форму кнопку и установите свойству (**Name**) значение `AddRowButton`, свойству **Text** – *Add Row.*
12. Для новой кнопки создайте обработчик события `Click` и реализуйте следующую функциональность:
    a. Создайте новый экземпляр строки `Customers`:

```
NorthwindDataSet.CustomersRow NewRow =
    (NorthwindDataSet.CustomersRow)northwindDataSet1.Customers.NewRow();
```

b. Присвойте значения каждому столбцу строки:

```
NewRow.CustomerID = "WINGT";
NewRow.CompanyName = "Wingtip Toys";
NewRow.ContactName = "Steve Lasker";
NewRow.ContactTitle = "CEO";
NewRow.Address = "1234 Main Street";
NewRow.City = "Buffalo";
NewRow.Region = "NY";
NewRow.PostalCode = "98052";
NewRow.Country = "USA";
NewRow.Phone = "206-555-0111";
NewRow.Fax = "206-555-0112";
```

c. Реализуйте добавление строки к коллекции `Rows` таблицы `Customers`:

```
try
   {
    northwindDataSet1.Customers.Rows.Add(NewRow);
   }

catch (Exception ex)
   {
    MessageBox.Show(ex.Message, "Add Row Failed");
   }
```

13. Добавьте к классу формы метод, возвращающий выбранный в сетке `CustomersRow`:

```
private NorthwindDataSet.CustomersRow GetSelectedRow()
     {
      String SelectedCustomerID =
CustomersDataGridView.CurrentRow.Cells["CustomerID"].Value.ToString();
      NorthwindDataSet.CustomersRow SelectedRow =
northwindDataSet1.Customers.FindByCustomerID(SelectedCustomerID);
        return SelectedRow;
      }
```

14. Добавьте на форму кнопку и установите свойству (**Name**) значение `DeleteRowButton`, свойству **Text** – *Del Row*.
15. Для новой кнопки создайте обработчик события `Click` и реализуйте вызов метода `Delete()` выбранной строки для пометки ее в **DataTable** как удаленной:

```
GetSelectedRow().Delete();
```

16. Добавьте на форму три кнопки и установите следующие свойства:
    a. Первая:
        (**Name**) – `UpdateValueButton`,
        Text – *Update Value*
    b. Вторая:
        (**Name**) – `AcceptChangesButton`,
        Text – *Accept Changes*
    c. Третья:
        (**Name**) – `RejectChangesButton`,
        Text – *Reject Changes*
17. Добавьте четыре текстовых поля TexBox и присвойте свойствам (**Name**) следующие значения: `CellValueTextBox`, `OriginalDRVTextBox`, `CurrentDRVTextBox`, `RowStateTextBox`
18. Добавьте к классу формы метод, обновляющий текстовые поля версиями и состояниями строки:

```
private void UpdateRowVersionDisplay()
   {
       }
```

19. В этом методе отобразите значения `Original` и `Current` для `DataRowVersion` выбранной `Cell`:

```
try
            {
    CurrentDRVTextBox.Text =
GetSelectedRow()[CustomersDataGridView.CurrentCell.OwningColumn.Name,
DataRowVersion.Current].ToString();
            }
catch (Exception ex)
            {
                CurrentDRVTextBox.Text = ex.Message;
            }
try
            {
                OriginalDRVTextBox.Text =
GetSelectedRow()[CustomersDataGridView.CurrentCell.OwningColumn.Name,
DataRowVersion.Original].ToString();
            }
```

```
catch (Exception ex)
            {
                OriginalDRVTextBox.Text = ex.Message;
            }
```

20. В этом же методе отобразите текущий `RowState` выбранной строки:

```
RowStateTextBox.Text = GetSelectedRow().RowState.ToString();
```

21. Создайте обработчик события `Click` для кнопки `UpdateValueButton` и добавьте следующий код:

```
GetSelectedRow()[CustomersDataGridView.CurrentCell.OwningColumn.Name] =
CellValueTextBox.Text;
UpdateRowVersionDisplay();
```

22. Создайте обработчик события `CustomersDataGridView_Click` и добавьте следующий код:

    a. Заполните `CellValueTextBox` значением выбранной ячейки

```
CellValueTextBox.Text =
CustomersDataGridView.CurrentCell.Value.ToString();
```

    b. Обновите другие текстовые поля:

```
UpdateRowVersionDisplay();
```

23. Создайте обработчик события `AcceptChangesButton_Click` и добавьте следующий код:

```
GetSelectedRow().AcceptChanges();
UpdateRowVersionDisplay();
```

24. Создайте обработчик события `RejectChangesButton_Click` и добавьте следующий код:

```
GetSelectedRow().RejectChanges();
UpdateRowVersionDisplay();
```

25. Запустите приложение и нажмите кнопку *Fill Table*. Щелкните в сетке и обратите внимание, что значения `Original` и `Current` показывают одинаковые значения, а `RowState` показывает *`Unchanged`*.
26. Щелкните ячейку, содержащую *Maria Anders* (первая строка) и введите *Maria AndersEdited* в `CellValueTextBox`.
27. Обновите кнопкой *Update Value* и обратите внимание, что значение в сетке обновлено, текстовые поля `Original` и `Current` отображают различные версии записи, а текст в `RowState` изменен на `Modified`.
28. Щёлкните кнопку *Add Row*.

29. Пролистайте сетку вниз и выделите одну из ячеек в новой записи (WINGT). Обратите внимание, что *Row State* содержит Added, указывающее, что это новая строка, а текстовое поле `Original` показывает отсутствие исходных данных.
30. Пролистайте сетку назад до строки с полем *MariaAndersEdited* и выделите его.
31. Щёлкните кнопку *Reject Changes* и проверьте значения версий и состояния строки.
32. Пролистайте до записи WINGT и выделите ее. Щёлкните кнопку *Accept Changes* и проверьте значения версий и состояния строки.

Упражнение 6. Создание и использование объектов DataView

В этом упражнении рассматривается использование объектов **DataView**, которые позволяют работать с объектами **DataTable** и обеспечивают возможность сортировки, фильтрации и изменения данных в связанном объекте **DataTable**.

1. Создайте новое WinForms-приложение (назовите его DataViewExample).
2. На поверхность формы проекта перенесите ЭУ **DataGridView**, свойству (**Name**) укажите значение `CustomersGrid`. Сделайте так, чтобы компонент занимал левую половину формы.
3. Добавьте на форму два элемента **TextBox** и установите следующие свойства:
    a. Первый:

        (**Name**) – `SortTextBox`,

        Text – *CustomerID*

    b. Второй:

        (**Name**) – `FilterTextBox`,

        Text – *City = 'London'*
4. Добавьте на форму две кнопки и установите следующие свойства:
    a. Первая:

        (**Name**) – `SetDataViewPropertiesButton`,

        Text – *SetDataViewProperties*

    b. Вторая:

        (**Name**) – `AddRowButton`,

        Text – *Add Row*

5. Создайте новый источник данных с помощью команды **Add New Data Source** (Добавить новый источник данных) в меню **Data** (Данные):
    a. Тип источника – **Database**,
    b. Модель данных – **DataSet** (Набор данных),
    c. Подключение – Northwind,
    d. Объекты базы данных – таблицы `Customers` и `Orders`.
    e. Имя источника оставьте по умолчанию.
6. Постройте проект. Проверьте, что на панели **Toolbox** появились компоненты **CustomersTableAdapter**, **OrdersTableAdapter** и **NorthwindDataSet**. Перенесите все эти компоненты на форму.
7. В классе формы создайте экземпляры **DataView** для таблиц `Customers` и `Orders`:

```
DataView customersDataView;
DataView ordersDataView;
```

8. Создайте обработчик события загрузки формы и реализуйте следующую функциональность:
    a. Загрузите данные в таблицы `Customers` и `Orders`:

```
customersTableAdapter1.Fill(northwindDataSet1.Customers);
ordersTableAdapter1.Fill(northwindDataSet1.Orders);
```

    b. Настройте объекты **DataSet** для использования таблиц `Customers` и `Orders`:

```
customersDataView = new DataView(northwindDataSet1.Customers);
ordersDataView = new DataView(northwindDataSet1.Orders);
```

    c. Присвойте исходный порядок сортировки в **DataView**:

```
customersDataView.Sort = "CustomerID";
```

    d. Настройте **CustomerGrid** для отображения **CustomerDataView**:

```
CustomersGrid.DataSource = customersDataView;
```

9. Запустите приложение. При загрузке формы данные **CustomerDataView** отобразятся в сетке. Закройте приложение.
10. В обработчике события `Click` кнопки **Set Data View** реализуйте сортировку и фильтрацию данных в зависимости от значений, введенных в элементы `SortTextBox` и `FilterTextBox` соответственно:

```
customersDataView.Sort = SortTextBox.Text;
customersDataView.RowFilter = FilterTextBox.Text;
```

11. Запустите приложение и протестируйте добавленную функциональность. Отобразите данные, соответствующие городу *City = 'Madrid'*.

12. Реализуйте в обработчике события `Click` кнопки *Add Row* следующую функциональность:
    a. Создайте новую строку:
    ```
    DataRowView newCustomRow = customersDataView.AddNew();
    ```
    b. Присвойте значения столбцам `CustomerID` и `CompanyName` значения:
    ```
    newCustomRow["CustomerID"] = "WINGT";
    newCustomRow["CompanyName"] = "Wing Tip Toys";
    ```
    c. Укажите явное окончание редактирования:
    ```
    newCustomRow.EndEdit();
    ```
13. Добавьте еще одну кнопку (например, под кнопкой *Add Row*) и установите свойству (**Name**) значение `GetOrdersButton`, свойству **Text** – *Get Orders.*
14. Добавьте новый компонент **DataGridView** так, чтобы он занимал правую половину формы, свойству (**Name**) укажите значение `OrdersGrid`.
15. Реализуйте в обработчике события `Click` кнопки *Get Orders* следующую функциональность:
    a. Получите `CustomerID` для строки, выбранной в `CustomersGrid`:
    ```
    string selectedCustomerID =
(string)CustomersGrid.SelectedCells[0].OwningRow.Cells["CustomerID"].Value;
    ```
    b. Создайте **DataRowView** и присвойте ему выбранную строку:
    ```
    DataRowView selectedRow =
  customersDataView[customersDataView.Find(selectedCustomerID)];
    ```
    c. Вызовите метод `CreateChildView()` для перемещения по записям и создания нового **DataView**, основанного на связанных записях:
    ```
    ordersDataView =
  selectedRow.CreateChildView(northwindDataSet1.Relations
                                  ["FK_Orders_Customers"]);
    ```
    d. Настройте `OrderGrid` для отображения связанного **DataView**:
    ```
    OrdersGrid.DataSource = ordersDataView;
    ```
16. Запустите и протестируйте приложение. Выберите клиента (`CustomerID`) и просмотрите его заказы.

Практическое занятие 5. Создание элементов управления, связанных с данными

Windows Forms предоставляет классы для связывания данных между элементом управления, например TextBox, и данными из источника. Когда

элементы управления на форме связаны с данными, базовый источник данных синхронизируется со значениями в элементах управления. Связывание данных позволяет доставить их в приложение, обычно в набор данных, а затем изменить эти данные, редактируя значения в связанных с данными элементах и отправляя изменения обратно в базу данных.

Упражнение 1. Создание связанной с данными формы с помощью мастера

Мастер **Data Source Configuration Wizard** создает в приложении типизированный **DataSet** и заполняет окно **Data Sources** объектами, выбранными во время работы мастера. В этом упражнении создается форма, отображающая данные в **DataGridView**.

1. Создайте новое WinForms-приложение (назовите его DataSourcesWizard).
2. В меню **Data** выберите команду **Add New Data Sources** (Добавить новый источник данных) и запустите мастер настройки источника данных.
3. Настройте источник подключения к базе данных Northwind с набором данных: таблицы `Customers`, `Order Details`, `Orders`.
4. В меню **Data** (Данные) выберите команду **Show Data Sources** (Показать источники данных).
5. Перенесите узел `Customers` на форму. Изучите компоненты, которые добавились на форму и в область компонентов.
6. Запустите приложение. В итоге получилось работающее приложение с **DataGridView**, связанным с данными таблицы `Customers`.
7. В окне **Data Sources** (Источники данных) разверните узел `Customers`. Перенесите на форму узел `Orders`, **вложенный** в узел `Customers`. Обратите внимание на `OrdersBindingSource` и `OrdersTableAdapter`, добавленные в область компонентов.
8. Запустите приложение. Щелкните строку в таблице `Customers`. Обратите внимание, что `OrdersDataGridView` отображает все заказы выбранного клиента.

Упражнение 2. Связывание данных с элементами управления

В этом упражнении описывается, как реализовать простое связывание элементов управление Windows Forms с данными и перемещаться вперед и назад по записям данных.

1. Создайте новое WinForms-приложение (назовите его DataBindingSimple).

2. В меню **Data** выберите команду **Add New Data Sources** (Добавить новый источник данных) и запустите мастер настройки источника данных.
3. Настройте источник подключения к базе данных Northwind с набором данных: таблица `Products`.
4. Постройте проект.
5. Перенесите **NorthwindDataSet** и **ProductsDataAdapter** на форму.
6. Добавьте на форму два элемента **TextBox** и установите свойствам (**Name**) для первого элемента значение `ProductIDTextBox`, для второго – `ProductNameTextBox`.
7. Добавьте на форму две кнопки для перемещения по записям и установите им следующие свойства:
    c. Первая:

        (**Name**) – `PreviousButton`,

        Text – *Previous*

    d. Вторая:

        (**Name**) – `NextButton`,

        Text – *Next*
8. В классе формы объявите экземпляр класса `BindingSource`:

```
private BindingSource productsBindingSource;
```

9. Создайте обработчик события загрузки формы и реализуйте следующую функциональность:
    a. Загрузите данные в таблицу `Products`:

```
productsTableAdapter1.Fill(northwindDataSet1.Products);
```

    b. Создайте `BindingSource` для таблицы `Products`:

```
productsBindingSource = new BindingSource(northwindDataSet1, "Products");
```

    c. Настройте связывание для **TextBox**:

```
ProductIDTextBox.DataBindings.Add("Text",
productsBindingSource, "ProductID");
ProductNameTextBox.DataBindings.Add("Text",
productsBindingSource, "ProductName");
```

10. Реализуйте в обработчике события `Click` кнопки *Previous* перемещение к предыдущей записи в источнике данных:

```
productsBindingSource.MovePrevious();
```

11. Реализуйте в обработчике события `Click` кнопки *Next* перемещение к следующей записи в источнике данных:

```
productsBindingSource.MoveNext();
```

12. Запустите приложение и протестируйте работу кнопок. Каждое тестовое поле связано с одним столбцом в таблице данных и объект управляет обслуживанием низкого уровня.

Упражнение 3. Сложное связывание данных

В упражнении показывается, как выполнить сложное связывание с данными элемента управления (**DataGridView**). В этом упражнении связывание данных настраивается в коде. В предыдущих упражнениях связывание реализовывалось перемещением требуемых элементов из окна **Data Source**.

1. Создайте новое WinForms-приложение (назовите его DataBindingComplex).
2. В меню **Data** выберите команду **Add New Data Sources** (Добавить новый источник данных) и запустите мастер настройки источника данных.
3. Настройте источник подключения к базе данных Northwind с набором данных: таблица `Products`.
4. Постройте проект.
5. Перенесите **NorthwindDataSet** и **ProductsDataAdapter** на форму.
6. Перенесите **DataGridView** на форму, свойству (**Name**) установите значение `ProductsGrid`. Сделайте так, чтобы компонент занимал верхнюю половину формы.
7. Перенесите на форму **BindingNavigator**.
8. Добавьте на форму кнопку и установите свойству (**Name**) значение `BindGridButton`, свойству **Text** – *Bind Grid*.
9. Реализуйте в обработчике события `Click` кнопки *Bind Grid* следующую функциональность:
    a. Заполните таблицу `Products` данными из базы данных:
    ```
    productsTableAdapter1.Fill(northwindDataSet1.Products);
    ```
    b. Создайте новый компонент `BindingSource` для таблицы `Products`:
    ```
    BindingSource productsBindingSource = new
BindingSource(northwindDataSet1, "Products");
    ```
    c. Свяжите сетку с компонентом `BindingSource`:
    ```
    ProductsGrid.DataSource = productsBindingSource;
    ```
10. Свяжите навигатор с компонентом `BindingSource`:
    ```
    bindingNavigator1.BindingSource = productsBindingSource;
    ```
11. Запустите приложение и загрузите в сетку данные. Протестируйте работу навигатора, настроенного для использования того же объекта, что и сетка.

Упражнение 4. Работа с DataGridView

В этом упражнении более подробно рассматривается элемент управления – сетка **DataGridView**.

1. Создайте новое WinForms-приложение (назовите его DataGridViewExample).
2. В меню **Data** выберите команду **Add New Data Sources** (Добавить новый источник данных) и запустите мастер настройки источника данных.
3. Настройте источник подключения к базе данных Northwind с набором данных: таблица `Customers`.
4. Перенесите узел `Customers` из окна `Data Source` на форму.
5. Откройте созданный конструктором обработчик события `Load` формы, посмотрите, что уже реализована загрузка данных в таблицу `Customers` и укажите код для создания нового столбца в **DataTable**:

```
DataColumn Location = new DataColumn("Location");
Location.Expression = "City + ',' + Country";
northwindDataSet.Customers.Columns.Add(Location);
```

6. Добавьте на форму две кнопки и установите следующие свойства:
   e. Первая:

      (**Name**) – `AddColumnButton`,

      Text – *Add Column*

   f. Вторая:

      (**Name**) – `DeleteColumnButton`,

      Text – *Delete Column*

7. Реализуйте в обработчике события `Click` кнопки *Add Column* добавление нового столбца в **DataGridView:**

```
DataGridViewTextBoxColumn LocationColumn = new
DataGridViewTextBoxColumn();
LocationColumn.Name = "LocationColumn";
LocationColumn.HeaderText = "Location";
LocationColumn.DataPropertyName = "Location";
customersDataGridView.Columns.Add(LocationColumn);
```

8. Реализуйте в обработчике события `Click` кнопки *Delete Column* удаление столбца в **DataGridView:**

```
try
{
```

```
            customersDataGridView.Columns.Remove("LocationColumn");
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
```

9. Добавьте на форму еще две кнопки и установите следующие свойства:
    a. Первая:

        (**Name**) – `GetClickedCellButton`,

        Text – *Get Clicked Cell*

    b. Вторая:

        (**Name**) – `ApplyStyleButton`,

        Text – *Apply Style*

10. Добавьте на форму рядом с кнопкой *Get Clicked Cell* ЭУ Label.
11. Реализуйте в обработчике события `Click` кнопки *Get Clicked Cell* следующую функциональность:
    a. Объявите переменную для хранения информации о выбранной ячейке:

```
string CurrentCellInfo;
```

    b. Присвойте объявленной переменной содержимое ячейки:

```
CurrentCellInfo =
customersDataGridView.CurrentCell.Value.ToString() + Environment.NewLine;
```

    c. Добавьте к переменной информацию об имени столбца, а также индексах столбца и строки:

```
CurrentCellInfo += "Column: " +
  customersDataGridView.CurrentCell.OwningColumn.DataPropertyName +
Environment.NewLine;
CurrentCellInfo += "Column Index: " +
    customersDataGridView.CurrentCell.ColumnIndex.ToString() +
Environment.NewLine;
CurrentCellInfo += "Row Index: " +
    customersDataGridView.CurrentCell.RowIndex.ToString() +
Environment.NewLine;
```

    d. Результирующую строку выведете в поле надписи формы:

```
label1.Text = CurrentCellInfo;
```

12. Добавьте обработчик события `CellValidating` компонента **DataGridView** и реализуйте проверку на соответствие правилам ввода, например, чтобы поле `ContactName` содержало некоторое значение:

```
            if
(customersDataGridView.Columns[e.ColumnIndex].DataPropertyName ==
"ContactName")
            {
                if (e.FormattedValue.ToString() == "")
                {
                    customersDataGridView.Rows[e.RowIndex].ErrorText =
                        "ContactName is a required field";
                    e.Cancel = true;
                }
                else
                    customersDataGridView.Rows[e.RowIndex].ErrorText = "";
            }
```

13. Реализуйте в обработчике события `Click` кнопки *Apply Style* чередующиеся строки со светло-серым фоном:

```
  customersDataGridView.AlternatingRowsDefaultCellStyle.BackColor =
Color.LightGray;
```

14. Запустите и протестируйте приложение. Добавьте столбец Location, проверьте добавление столбца. Удалите столбец. Получите информацию о ячейке с помощью кнопки *Get Clicked Cell*. Примените отображение чередующихся строк со светло-серым фоном.

Практическое занятие 6. Работа с XML в объектах DataSet

Объект **DataSet** модели ADO.NET изначально рассчитан на работу с XML. Содержимое **DataSet** можно загружать и сохранять в виде XML-документов. Кроме того, **DataSet** позволяет выделить информацию схемы (сведения о таблицах, столбцах и ограничениях) в файл XML-схемы. Для работы с данными XML объекты **DataSet** имеют несколько методов, которые рассматриваются в следующих упражнениях.

Упражнение 1. Сохранение объектов DataSet как XML.

В этом упражнении данные из **DataSet** сохраняются отформатированными в XML-файле.

1. Создайте новое WinForms-приложение (назовите его SavingDataSetXml).

2. Перенесите на форму компонент **SqlDataAdapter**, откроется мастер настройки адаптера. На первой странице укажите подключение к БД Northwind.
3. При выборе типа команды оставьте по умолчанию – использовать инструкции SQL.
4. Введите инструкцию `SELECT * FROM Customers`. Проверьте, что в области компонентов появился объект **Connection**.
5. Измените свойство (**name**) `SqlAdapter1` на `CustomersAdapter`.
6. Перенесите на форму второй компонент **SqlDataAdapter**, снова откроется мастер настройки адаптера.
7. При выборе типа команды оставьте по умолчанию – использовать инструкции SQL.
8. Введите инструкцию `SELECT * FROM Orders`.
9. Измените свойство (**name**) `SqlAdapter1` на `OrdersAdapter`.
10. Сгенерируйте типизированный **DataSet**, основанный на настроенных адаптерах, для чего в меню **Data** (Данные) выберите команду **Generate DataSet** (Создать набор данных). Обратите внимание, что выбраны настроенные адаптеры. Для нового набора данных введите имя **NorthwindDataSet**.
11. Добавьте на форму **DataGridView**. Сделайте так, чтобы компонент занимал верхнюю половину формы. Измените его свойство (**Name**) на `CustomersGrid`.
12. Добавьте на форму три кнопки и установите следующе свойства:
    a. Первая:

       (**Name**) – `FillDataSetButton`,

       Text – *Fill*

    b. Вторая:

       (**Name**) – `SaveXmlDataButton`,

       Text – *Save XML Data*

    c. Третья:

       (**Name**) – `SaveXmlSchemaButton`,

       Text – *Save XML Schema*
13. Создайте обработчик события `FillDataSetButton_Click` и реализуйте следующую функциональность:
    a. Заполните таблицы `Cusromers` и `Orders`:

```
CustomersAdapter.Fill(northwindDataSet1.Customers);
```

```
OrdersAdapter.Fill(.Orders);
```

b. Свяжите сетку с таблицей `Cusromers`:

```
CustomersGrid.DataSource = northwindDataSet1.Customers;
```

14. Создайте обработчик события `SaveXmlDataButton_Click` и сохраните данные `northwindDataSet1` в файл XML:

```
try
{
    northwindDataSet1.WriteXml("Northwind.xml");
    MessageBox.Show("Data save as XML");
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

15. Создайте обработчик события `SaveXmlSchemaButton_Click` и сохраните данные схемы объекта `northwindDataSet1` в файл XML:

```
try
{
    northwindDataSet1.WriteXmlSchema("Northwind.xsd");
    MessageBox.Show("Schema save as XML");
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

16. Запустите приложение. Загрузите в таблицу данные, сохраните данные и схему в соответствующие файлы. Проверьте, что в папке Debug приложения присутствуют сохраненные файлы Northwind.xml и Northwind.xsd. Эти файлы потребуются в следующем упражнении.

Упражнение 2. Загрузка объектов DataSet данными XML

В этом упражнении создается не типизированный **DataSet** и определяется его схема, основанная на содержимом файла Northwind.xsd. После загрузки схемы **DataSet** загружается содержимым файла Northwind.xml и отображается в сетке.

1. Создайте новое WinForms-приложение (назовите его LoadDataSetXml).
2. Скопируйте файлы Northwind.xml и Northwind.xsd, созданные в прошлом упражнении в папку Debug текущего проекта.

3. Добавьте на форму два компоненте **DataGridView**. Разместите их по своему усмотрению. Укажите их свойствам (**Name**) `CustomersGrid` и `OrdersGrid` соответственно.
4. Добавьте на форму две кнопки и установите следующе свойства:
    a. Первая:

        (**Name**) – `LoadSchemaButton`,

        Text – *Load Schema*

    b. Вторая:

        (**Name**) – `LoadDataButton`,

        Text – *Load Data*

5. Создайте нетипизированный объект **DataSet**:

```
DataSet NorthwindDataSet = new DataSet();
```

6. Создайте обработчик события `LoadSchemaButton_Click` и реализуйте следующую функциональность:
    a. Загрузите сведения схемы из файла .xsd:

        ```
        NorthwindDataSet.ReadXmlSchema("Northwind.xsd");
        ```

    b. Свяжите `CustomersGrid` и `OrdersGrid` для отображения данных:

    ```
    CustomersGrid.DataSource = NorthwindDataSet.Tables["Customers"];
    OrdersGrid.DataSource = NorthwindDataSet.Tables["Orders"];
    ```

7. Создайте обработчик события `LoadDataButton_Click` и загрузите данные в набор данных:

    ```
    NorthwindDataSet.ReadXml("Northwind.xml");
    ```

8. Запустите приложение и загрузите схему. Сетки должны отобразить столбцы соответствующих таблиц. Загрузите данные. Убедитесь, что данные отображаются в соответствующих сетках формы.

LINQ

Практическое занятие 7. Создание запросов на языке C# (LINQ)

В упражнениях этого занятия реализуются возможности языка C#, которые служат для написания выражений запросов LINQ.

После выполнения этого упражнения можно переходить к работе с конкретными поставщиками LINQ, например, LINQ to SQL, LINQ в DataSet или LINQ to XML.

Создание проекта

1. Создайте консольное приложение, назовите его Linq_Student.
2. Обратите внимание, что проект содержит ссылку на **System.Core.dll** и директиву **using** для пространства имен System.Linq.

Создание расположенного в памяти источника данных

Источником данных для запросов в этом упражнении является список объектов `Student`. Каждая запись `Student` имеет имя, фамилию и массив целых чисел, представляющий результаты тестирования.

1. Добавьте класс `Student`.
2. Добавьте инициализированный список учащихся (создайте несколько студентов по обзазцу):

```
public class Student
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public List<int> Scores;
}

static List<Student> students = new List<Student>
{
   new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new
List<int> {97, 92, 81, 60}},
    new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new
List<int> {75, 84, 91, 39}},
};
```

3. Обратите внимание на то что класс `Student` состоит из автоматически реализованных свойств, каждый учащийся в списке инициализируется с помощью инициализатора объектов, а сам список инициализируется с помощью инициализатора коллекцией.

Упражнение 1. Создание и выполнение простого запроса

1. В методе **Main** создайте простой запрос, который при выполнении вернет список всех учащихся, результат тестирования которых, по первой оценке, превысил 90 баллов.

Указание. Поскольку объект `Student` выбирается целиком, типом запроса укажите `IEnumerable<Student>`:

```
IEnumerable<Student> studentQuery =
```

```
    from student in students
    where student.Scores[0] > 90
    select student;
```

2. Реализуйте цикл foreach, который приведет к выполнению запроса:

```
foreach (Student student in studentQuery)
{
    Console.WriteLine("{0}, {1}", student.Last, student.First);
}
```

Обратите внимание на следующие моменты.

- Доступ к каждому элементу в возвращаемой последовательности осуществляется с помощью переменной итерации в цикле `foreach`.
- Типом этой переменной является `Student`, а типом переменной запроса является совместимый IEnumerable<Student>.

3. Выполните построение и запустите приложение.

Упражнение 2. Изменение запроса

Добавление дополнительного условия фильтра

Чтобы уточнить запрос, можно объединить несколько логических условий в предложении `where`.

1. В предыдущий запрос добавьте условие таким образом, чтобы запрос возвращал тех учащихся, первый результат которых был более 90 баллов, а последний результат был меньше 80.

Предложение `where` должно выглядеть следующим образом:

```
where student.Scores[0] > 90 && student.Scores[3] < 80
```

Упорядочение результатов

Просматривать результаты легче, если они как-то упорядочены. Возвращаемую последовательность можно упорядочить по любому доступному полю в исходных элементах.

1. Добавьте следующее предложение `orderby` к запросу, указав его после инструкции `where` и перед оператором `select`:

```
orderby student.Last ascending
```

2. Измените предложение `orderby` таким образом, чтобы оно сортировало результаты в обратном порядке по результату первого тестирования, от высшего к низшему показателю:

```
orderby student.Scores[0] descending
```

3. Измените строку форматирования WriteLine, чтобы видеть результаты тестирования:

```
Console.WriteLine("{0}, {1} {2}", student.Last, student.First,
student.Scores[0]);
```

Группировка результатов

Запрос с предложением group создает последовательность групп, где каждая группа содержит `Key` и последовательность, состоящую из всех членов этой группы.

1. Создайте новый запрос, который группирует учащихся по первой букве их фамилии в качестве ключа:

```
var studentQuery2 =
    from student in students
    group student by student.Last[0];
```

Обратите внимание, что тип запроса изменился. Теперь он создает последовательность из групп, имеющих тип `char` в качестве ключа, и последовательность объектов `Student`.

2. Поскольку тип запроса изменился, измените также и цикл foreach:

```
foreach (var studentGroup in studentQuery2)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine("   {0}, {1}",
                  student.Last, student.First);
    }
}
```

3. Запустите и протестируйте приложение.

Упражнение 3. Расширение возможностей запроса

Присвоение переменной неявного типа

Явное кодирование **IEnumerables** из **IGroupings** может быстро стать трудоемким.

С помощью `var` можно более просто написать тот же запрос и цикл foreach. Ключевое слово var не приводит к изменению типов объектов; оно просто сообщает компилятору о необходимости определения типов.

1. Создайте новый запрос и обратите внимание на применение ключевого слова var:

```
var studentQuery3 =
    from student in students
    group student by student.Last[0];

foreach (var groupOfStudents in studentQuery3)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
         Console.WriteLine("   {0}, {1}",
             student.Last, student.First);
    }
}
```

Упорядочение групп по значению их ключа

При выполнении предыдущего запроса группы были расположены не в алфавитном порядке.

2. Создайте новый запрос на основе предыдущего.
3. Укажите предложение `orderby` после предложения `group`.
4. Чтобы использовать предложение `orderby`, нужен идентификатор, служащий в качестве ссылки на группы, создаваемые предложением `group`. Предоставьте идентификатор с помощью ключевого слова `into` следующим образом:

```
var studentQuery4 =
    from student in students
    group student by student.Last[0] into studentGroup
    orderby studentGroup.Key
    select studentGroup;

foreach (var groupOfStudents in studentQuery4)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine("   {0}, {1}",
            student.Last, student.First);
```

```
        }
    }
```

5. Постройте и запустите приложение. При выполнении этого запроса группы будут отсортированы в алфавитном порядке.

Введение идентификаторов с помощью let

Ключевое слово `let` можно использовать для представления идентификатора для любого результата выражения в выражении запроса.

6. Создайте новый запрос:

```
var studentQuery5 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where totalScore / 4 < student.Scores[0]
    select student.Last + " " + student.First;

foreach (string s in studentQuery5)
{
    Console.WriteLine(s);
}
```

7. Исследуйте работу запроса. Обратите внимание, что идентификатор `let` применяется больше для удобства, хотя в других случаях он может повысить производительность, сохраняя результаты выражения так, чтобы оно не вычислялось повторно.

Использование синтаксиса метода в выражении запроса

Некоторые операции запроса могут быть выражены только с помощью синтаксиса методов.

8. Создайте новый запрос в котором вычисляется общий результат для каждого Student в исходной последовательности, а затем вызовите метод **Average()**, использующий результаты запроса для вычисления среднего балла класса. Обратите внимание на круглые скобки вокруг выражения запроса.

```
var studentQuery6 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    select totalScore;
```

```
    double averageScore = studentQuery6.Average();
Console.WriteLine("Class average score = {0}", averageScore);
```

9. Постройте и запустите приложение.

Преобразование или проецирование в предложении select

Очень часто в запросах создаются последовательности, элементы которых отличаются от элементов в исходных последовательностях.

10. Создайте новый запрос, который возвращает последовательность строк (не Students):

```
IEnumerable<string> studentQuery7 =
    from student in students
    where student.Last == "Garcia"
    select student.First;
```

11. В цикле foreach реализуйте перебор строк:

```
Console.WriteLine("The Garcias in the class are:");
foreach (string s in studentQuery7)
{
    Console.WriteLine(s);
}
```

12. Постройте и запустите приложение.
13. Создайте новый запрос на основе предыдущего.
14. Создайте последовательность `Students`, суммарный результат баллов которых больше среднего, вместе с их `StudentID`, используйте анонимный тип в инструкции `select`:

```
var studentQuery8 =
    from student in students
    let x = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where x > averageScore
    select new { id = student.ID, score = x };


foreach (var item in studentQuery8)
{
    Console.WriteLine("Student ID: {0}, Score: {1}", item.id,
item.score);
}
```

15. Постройте и запустите приложение.

Практическое занятие 8. Использование LINQ to SQL

LINQ to SQL является удобным инструментом для разработчиков, которым необязательно сопоставление с концептуальной моделью. Использование **LINQ to SQL** дает возможность напрямую использовать модель программирования LINQ с существующей схемой базы данных. **LINQ to SQL** позволяет разработчикам формировать классы .NET Framework для представления данных. Вместо сопоставления с концептуальной моделью эти созданные классы указывают напрямую на таблицы базы данных, представления, хранимые процедуры и определяемые пользователем функции.

В **LINQ to SQL** объектная модель, выраженная на языке программирования разработчика, сопоставляется модели данных реляционной базы данных. После этого операции с данными выполняются в соответствии с объектной моделью.

Для создания объектной модели для базы данных, классы должны быть приведены в соответствие с сущностями, хранящимися в базе данных.

Можно выделить три способа реализации такого приведения: задавать атрибуты для существующих объектов самостоятельно, т.е. написать код для атрибута вручную, использовать конструктор, позволяющий автоматически сгенерировать объекты и использовать утилиту командной строки SQLMetal.

Упражнение 1. Создание объектной модели

В упражнении рассматривается способ создания объектной модели для базы данных: задание атрибутов для существующих объектов кодированием вручную.

1. Создайте новое WinForms-приложение (назовите его LINQsql_1).
2. На форму перенесите кнопку и **ListBox**.
3. Добавьте ссылку на сборку `System.Data.Linq` и директивы подключения требуемых пространств имен:

```
using System.Data.Linq;
using System.Data.Linq.Mapping;
```

4. Создайте класс сущностей (лучше в отдельном файле) `Customer` и для связывания его с таблицей базы данных к объявлению класса добавьте атрибут `Table`:

```
[Table(Name = "Customers")]
public class Customer
{
```

```
}
```

5. В классе назначьте свойства `CustomerID` и `City` для представления столбцов базы данных, для этого укажите атрибут `Column`:

```
private string _CustomerID;
[Column(IsPrimaryKey = true, Storage = "_CustomerID")]
public string CustomerID
{
    get
    {
        return this._CustomerID;
    }
    set
    {
        this._CustomerID = value;
    }
}

private string _City;
[Column(Storage = "_City")]
public string City
{
    get
    {
        return this._City;
    }
    set
    {
        this._City = value;
    }
}
```

6. Переопределите метод `ToString()`:

```
public override string ToString()
{
    return CustomerID + "\t" + City;
}
```

7. В обработчике события `Click` для кнопки реализуйте следующую функциональность:

a. Создайте объект класса `DataContext`, который задает входную точку в базу данных, и в качестве параметра конструктора укажите строку подключения к базе **Northwind**:

```
DataContext db = new DataContext
(@"Data Source=(local);Initial Catalog=Northwind;Integrated
Security=True");
```

b. Для объекта класса `DataContext` используйте метод `GetTable`, который возвращает коллекцию определенного типа, в данном случае типа `Customer` и составьте запрос для вывода в **ListBox** список идентификаторов и городов проживания тех заказчиков, которые живут в Лондоне:

```
var results = from c in db.GetTable<Customer>()
              where c.City == "London"
              select c;
foreach (var c in results)
listBox1.Items.Add(c.ToString());
```

8. Запустите и протестируйте приложение. По клику кнопки будут выведены идентификаторы и города проживания тех заказчиков, которые живут в Лондоне.

Упражнение 2. Создание связей в базе данных

В данном упражнении демонстрируется использование ассоциаций **LINQ to SQL** для представления связей внешних ключей в базе данных.

Будут решены следующие основные задачи:

- Добавление класса сущности, который представляет таблицу "`Orders`" в базе данных "**Northwind**".
- Добавление примечаний к классу `Customer`, чтобы расширить связи между классами `Customer` и `Order`.
- Создание и выполнение запроса для тестирования процесса получения сведений о классе `Order` с помощью класса `Customer`.

1. Откройте приложение, созданное в прошлом упражнении.
2. После определения класса `Customer` (лучше в отдельном файле) создайте определение класса сущностей `Order`, включающее следующий код, который указывает, что свойство `Order.Customer` связано как внешний ключ со свойством `Customer.CustomerID`:

```
[Table(Name = "Orders")]
public class Order
```

```
    {
        private int _OrderID = 0;
        private string _CustomerID;
        private EntityRef<Customer> _Customer;
        public Order() { this._Customer = new EntityRef<Customer>(); }

        [Column(Storage = "_OrderID", DbType = "Int NOT NULL IDENTITY",
        IsPrimaryKey = true, IsDbGenerated = true)]
        public int OrderID
        {
            get { return this._OrderID; }
            // No need to specify a setter because IsDBGenerated is
            // true.
        }

        [Column(Storage = "_CustomerID", DbType = "NChar(5)")]
        public string CustomerID
        {
            get { return this._CustomerID; }
            set { this._CustomerID = value; }
        }

        [Association(Storage = "_Customer", ThisKey = "CustomerID")]
        public Customer Customer
        {
            get { return this._Customer.Entity; }
            set { this._Customer.Entity = value; }
        }
    }
```

3. Если был создан отдельный файл, то добавьте директивы подключения требуемых пространств имен:

```
using System.Data.Linq;
using System.Data.Linq.Mapping;
```

4. Добавьте примечания к классу `Customer`, чтобы указать его связь с классом `Order`. Вставьте следующий код в класс `Customer`:

```
private EntitySet<Order> _Orders;
public Customer()
{
    this._Orders = new EntitySet<Order>();
```

```
}

[Association(Storage = "_Orders", OtherKey = "CustomerID")]
public EntitySet<Order> Orders
{
    get { return this._Orders; }
    set { this._Orders.Assign(value); }
}
```

5. Удалите компонент **ListBox** и добавьте компонент **ListView**.
6. Настройте **ListView** для дальнейшей работы: в свойстве `Columns` добавьте три столбца с заголовками (свойство `text`): *CustomerID, City и OrdersCount*, свойству `View` установите вид отображения `Details`.
7. В обработчике события `Click` кнопки измените запрос, и реализуйте вывод результата в таблицу компонента `listView`:

```
  var custQuery =
from cust in db.GetTable<Customer>()
where cust.Orders.Any()
select cust;

  foreach (var custObj in custQuery)
  {
      ListViewItem item =
          listView1.Items.Add(custObj.CustomerID.ToString());
      item.SubItems.Add(custObj.City.ToString());
      item.SubItems.Add(custObj.Orders.Count.ToString());
  }
```

8. Запустите и протестируйте приложение. По клику кнопки таблица заполнится данными о заказчиках и количество сделанных ими заказов.

Упражнение 3. Использование хранимых процедур

В данном упражнении рассматривается основной сценарий **LINQ to SQL** для получения доступа к данным, выполняя только хранимые процедуры. Этот метод часто используется администраторами баз данных для ограничения способов получения доступа к хранилищам данных.

В упражнении все обращения к хранилищу данных будут ограничены теми действиями, которые могут выполняться двумя хранимыми процедурами. Эти действия заключаются в возвращении продуктов, включенных в заказ с

введенным кодом, или истории продуктов, заказанных клиентом с введенным кодом.

Предварительное действие:

Для выполнения дальнейших действий требуется наличие файла кода C#, созданного из базы данных **Northwind**. Если такого файла нет, выполните команду `SqlMetal` со следующей командной строкой (перед выполнением команды отсоедините базу данных **Northwind**):

```
  sqlmetal /code:"c:\SQL Server 2000 Sample Databases\northwind.cs"
/language:csharp "c:\SQL Server 2000 Sample Databases\northwnd.mdf" /sprocs
/functions /pluralize
```

1. Создайте новое WinForms-приложение (назовите его LINQsqlSproc).
2. Добавьте ссылку на сборку `System.Data.Linq/`
3. Добавьте файл northwind.cs в проект.
4. Создайте подключения к базе данных – введите следующий код в класс формы (может быть другой путь к файлу данных):

```
 Northwnd db = new Northwnd(@"c:\SQL Server 2000 Sample
Databases\northwnd.mdf");
```

5. Перенесите две кнопки, два текстовых поля и две подписи с панели элементов на форму Form1.
6. Расположите элементы управления по своему усмотрению. При необходимости увеличьте размер формы, чтобы разместить все элементы управления.
7. Для первой подписи `label1` установите значение свойства `Текст` значение *Введите код заказа*, для второй подписи `label2` – *Введите код клиента*.
8. Для кнопок установите значение свойству `Текст`: *Подробности заказа* и *История заказа*.
9. Создайте обработчик события `Click` кнопки *Подробности заказа* и реализуйте следующую функциональность:
    a. Объявите переменную для хранения содержимого `textBox1` в качестве аргумента для хранимой процедуры:

```
 string param = textBox1.Text;
```

    b. Объявите переменную для хранения результатов возвращаемых хранимой процедурой `CustOrdersDetail`:

```
var custquery = db.CustOrdersDetail(Convert.ToInt32(param));
```

    c. Выполните хранимую процедуру и отобразите результаты:

```
string msg = "";
```

```
foreach (CustOrdersDetailResult custOrdersDetail in custquery)
{
    msg = msg + custOrdersDetail.ProductName + "\n";
}
if (msg == "")
    msg = "No results.";
MessageBox.Show(msg);
```

d. Очистите переменные для дальнейшего использования:

```
param = "";
textBox1.Text = "";
```

10. Создайте обработчик события `Click` кнопки *История заказа* и реализуйте функциональность подобную для кнопки *Подробности заказа* для хранимой процедуры `CustOrderHist`:

```
string param = textBox2.Text;
var custquery = db.CustOrderHist(param);
string msg = "";
  foreach (CustOrderHistResult custOrdHist in custquery)
   {
    msg = msg + custOrdHist.ProductName + "\n";
   }
MessageBox.Show(msg);
param = "";
textBox2.Text = "";
```

11. Запустите и протестируйте приложение:
    a. В поле *Введите код заказа* введите 10249 и нажмите кнопку *Подробности заказа*. В окне сообщения будет отображен список продуктов, включенных в заказ 10249.
    b. В поле *Введите код клиента* введите ALFKI и нажмите кнопку *История заказа*. Откроется окно сообщения, в котором отображается история заказа для клиента ALFKI.
    c. В поле *Введите код заказа* введите 123 и нажмите кнопку *Сведения о заказе*. Откроется окно сообщения, в котором отображается текст "Нет результатов".

Упражнение 4. Автоматическое создание объектов отображения с помощью Объектно-реляционного конструктора

В этом упражнении применяется реляционный конструктор объектов (Object Relational Designer) для автоматического создания объектной модели из метаданных существующей базы.

1. Создайте новое WinForms-приложение (назовите его LINQsql_m).
2. На форму перенесите кнопку (свойству `text` укажите значение *Десять лучших*) и **ListView**.
3. Настройте **ListView** для дальнейшей работы: в свойстве `Columns` добавьте два столбца с заголовками (свойство `text`): *Ten Most* и *UnitPrice*, свойству `View` установите вид отображения `Details`.
4. Добавьте в проект класс – файл **LINQ to SQL classes**, назовите его DataClassesProc.dbml.
5. В окне Server Explorer разверните дерево базы данных Northwind и перетащите хранимую процедуру `Ten Most Expensive Products`. Эта хранимая процедура извлекает из таблицы `Products` 10 самых дорогих продуктов и их цены.
6. В обработчике события `Click` кнопки *Десять лучших* укажите следующий код:

```
var db = new DataClassesProcDataContext();
foreach (var r in db.Ten_Most_Expensive_Products())
{
    ListViewItem item =
listView1.Items.Add(r.TenMostExpensiveProducts.ToString());
item.SubItems.Add(r.UnitPrice.ToString());
}
```

7. Обратите внимание, что входная точка в базу данных теперь создается конструктором класса `DataClassesProcDataContext` (в общем случае класс будет называться так: `{имя файла отображения} DataContext`), которому больше не нужно передавать параметром строку соединения в явном виде.
8. Запустите и протестируйте приложение. По клику кнопки 10 самых дорогих продуктов и их цены отобразятся в списке.

ADO.NET Entity Framework

Практическое занятие 9. Применение технологии ADO.NET Entity Framework

ADO.NET Entity Framework (EF) – объектно-ориентированная технология доступа к данным, являющееся *object-relational mapping* (ORM) решением для .NET Framework от Microsoft.

Платформа ADO.NET Entity Framework позволяет разработчикам создавать приложения для доступа к данным, работающие с концептуальной моделью приложения, а не напрямую с реляционной схемой хранения. Ее целью является уменьшение объема кода и усилий по обслуживанию приложений, ориентированных на обработку данных.

В данном практическом занятии рассматривается подходы **Code-First**, при использовании которого сначала определяется модель в коде, а затем, на ее основе создается (или модифицируется) база данных и **Database first,** при котором средства Visual Studio на основе созданной ранее базы данных формируют ее модель – Entity Data Model (EDM).

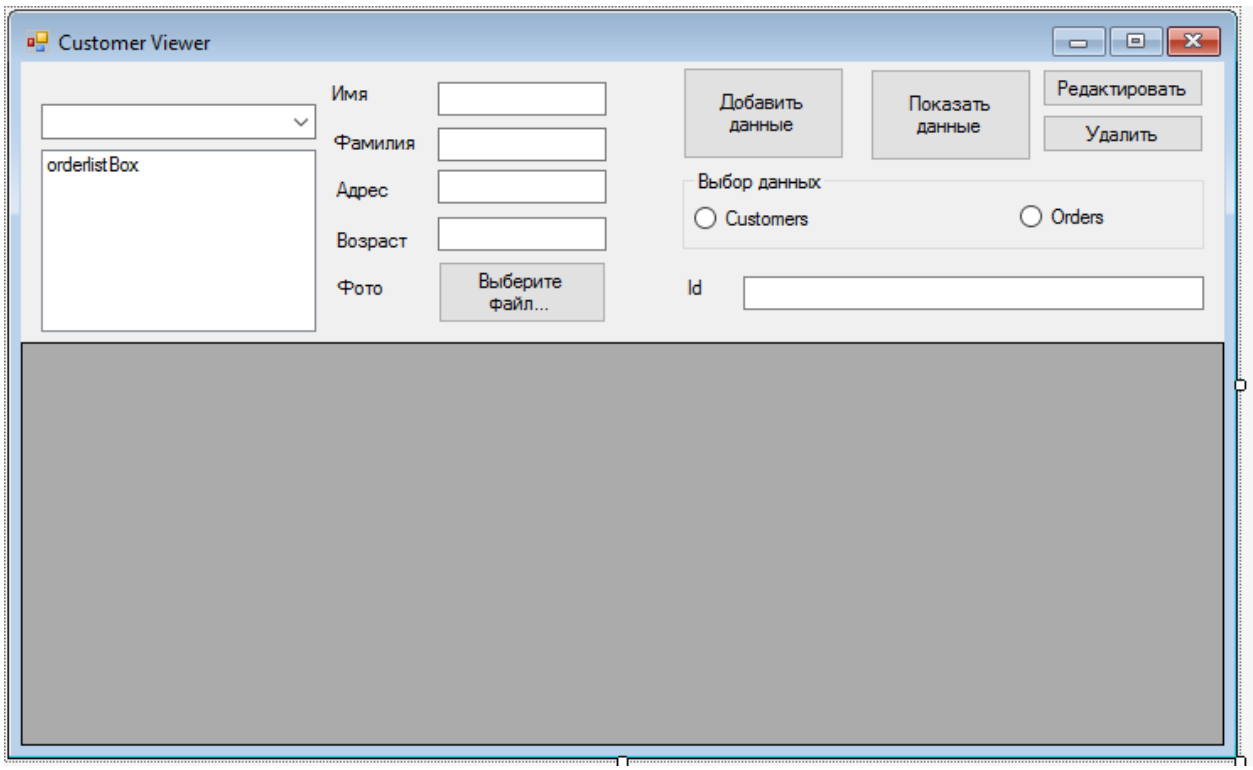
Платформа Entity Framework поддерживает модель EDM для определения данных на концептуальном уровне. При использовании конструктора ADO.NET EDM Designer концептуальная модель, модель хранения и сведения о сопоставлениях содержатся в файле EDMX.

Упражнение 1. Использование Code-First

В этом упражнении рассматривается подход **Code-First**, при использовании которого сначала определяется модель в коде, а затем, на ее основе создается (или модифицируется) база данных. Вы создадите две таблицы, описывающие данные заказчика и его заказы. Отношение между этими таблицами будет “один ко многим” (one-to-many).

Создание приложения

1. Создайте новое WinForms-приложение (назовите его **CustomerManager).**
2. Файлу формы присвойте имя **CustomerViewer.cs**.
3. Настройте свойства формы: значение свойства **(Name)** – `CustomerViewer`**,** значение свойства **Text** – `Customer Viewer`**.**
4. Реализуйте интерфейс приложения, изображенный на рисунке.



Интерфейс содержит следующие элементы:

- **ComboBox** (имя элемента `customerList`**)** для отображения заказчиков в виде списка
- **ListBox** (`orderlistBox`) для отображения списка заказов, для свойства **SelectedMode** укажите значение `MultiSimple` – возможность выбирать несколько элементов списка
- Элементы **TextBox** для ввода имени, фамилии, электронного адреса и возраста (в коде `textBoxname`, `textBoxlastname`, `textBoxmail` и `textBoxage` соответственно)
- Пять кнопок: для добавления фото заказчика (`buttonFile`), для добавления нового заказчика в базу (`buttonAdd`), для вывода информации из базы данных (`buttonOut`), для редактирования данных о заказчике (`buttonEdit`) и для удаления заказчика из базы данных (`buttonDel`)
- Два элемента **RadioButton**, размещенный в контейнере **GroupBox**, элементы позволят выводить данные о заказчиках (`CustomerradioButton`) или заказах (`OrderradioButton`)
- Элемент **TextBox** (`textBoxCustomer`) для вывода данных о заказчике и слева от него текстовая метка **Id** (`labelid`) для отображения уникального идентификатора заказчика

- Элемент **DataGridView** (`GridView`) для отображения данных, свойству **Dock** укажите значение `Botton`

5. Постройте приложение.

Создание модели данных

1. Добавьте в решение проект, имеющий шаблон библиотеки классов (Class Library) и назовите его **CodeFirst**.
2. Переименуйте файл класса Class1.cs, в файл Model.cs, в котором будет описана модель данных.
3. Добавьте ссылку на проект **CodeFirst** в базовом проекте приложения. Для этого щелкните правой кнопкой мыши по вкладке **References** (**Ссылки**) в окне **Solution Explorer** базового проекта и выберите пункт **Add Reference**. В открывшемся диалоговом окне перейдите на вкладку **Solution** и выберите проект **CodeFirst.**
4. Добавьте в файл Model.cs два класса, описывающих заказчика и его товары:

   ✓ Обратите внимание, что в классе **Customer** указано только поле имени, а на форме добавлено два текстовых поля для имени и фамилии, это правильно, так как в последствии модель будет изменена.

```
public class Customer
{
    public int CustomerId { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public int Age { get; set; }
    public byte[] Photo { get; set; }

    public override string ToString()
    {
        string s = Name + ", электронный адрес: " + Email;
        return s;
    }
    // Ссылка на заказы
    public virtual List<Order> Orders { get; set; }
    public Customer()
    {
        Orders = new List<Order>();
```

```
        }
    }

    public class Order
    {
        public int OrderId { get; set; }
        public string ProductName { get; set; }
        public string Description { get; set; }
        public int Quantity { get; set; }
        public DateTime PurchaseDate { get; set; }

        // Ссылка на покупателя
        public Customer Customer { get; set; }

        public override string ToString()
        {
            string s = ProductName + " " + Quantity + "шт., дата: " +
PurchaseDate;
            return s;
        }

    }
```

EF при работе с Code First требует определения ключа элемента для создания первичного ключа в таблице в БД. По умолчанию при генерации БД EF в качестве первичных ключей будет рассматривать свойства с именами `Id` или `[Имя_класса]Id` (т. е. `CustomerId`).

Код модели определяет данные заказчика и его покупки. Для каждого заказчика указывается имя, электронный адрес, возраст и фотография профиля. Идентификатор используется в качестве первичного ключа таблицы **Customer**. Кроме того, в этом классе есть ссылка на коллекцию покупок. Эта ссылка выражена в виде виртуального свойства и имеет тип обобщенной коллекции `List<T>`.

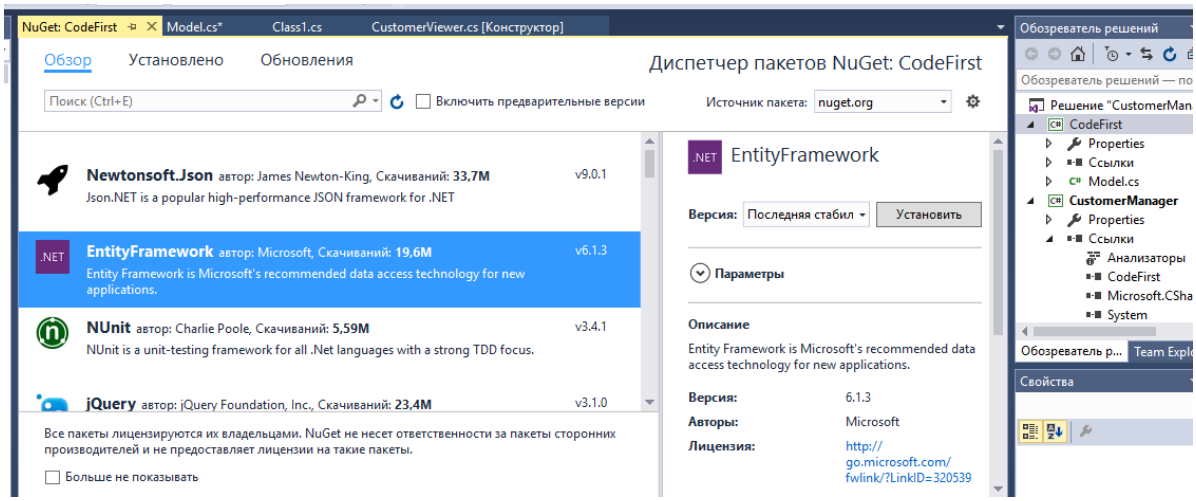
Класс **Order** содержит идентификатор заказа, который позволяет уникальным образом распознать каждый заказ в таблице. Кроме того, этот класс содержит свойства, описывающие название товара, его количество, описание и дату заказа. Также здесь указана ссылка на покупателя в виде свойства **Customer**.

Установка Entity Framework 6 в проект

Перед тем как использовать EF в проекте при подходе Code-First необходимо добавить ссылки на библиотеки EF.

Выполните следующие действия:

1. Для добавления поддержки EF с помощью **NuGet** (начиная с версии 4, библиотека EF входит в менеджер пакетов NuGet), выберите в окне **Solution Explorer** проект CodeFirst, щелкните по нему правой кнопкой мыши и выполните команду из контекстного меню **Manage Nuget Packages**.
2. В появившемся диалоговом окне на вкладке Обзор выберите последнюю версию **Entity Framework** и нажмите кнопку **Install** (Установить):



3. После этого появится окно с описанием лицензии на использование Entity Framework. Согласитесь с условиями, после чего NuGet установит в проект Entity Framework.
4. Добавьте ссылку на ключевое пространство имен в EF - System.Data.Entity в файле Model.cs:

```
using System.Data.Entity;
```

5. Повторите три первых действия для установки EF в проект CustomerManager.

Класс контекста данных

Созданные классы **Customer** и **Order** описывают структуру бизнес-модели, которая используется в приложении. Для того чтобы эти классы служили также для управления данными в базе данных, нужно использовать класс **контекста**.

EF имеет два базовых класса контекста: ***ObjectContext*** – этот класс является более общим классом контекста данных, и используется, начиная с самых

ранних версий EF и ***DbContext*** – этот класс контекста данных появился в Entity Framework 4.1, и он обеспечивает поддержку подхода Code-First (ObjectContext также обеспечивает работу подхода Code-First, но он труднее в использовании). Далее будет использоваться DbContext.

1. Для создания класса контекста добавьте новый класс **SampleContext** в проект **CodeFirst**.
2. В добавленном классе имя будущей базы данных укажите через вызов конструктора базового класса и укажите соответствующие таблицы (по традиции во множественном числе) базы данных с помощью свойств с типом `DbSet`:

```
using System.Data.Entity;
namespace CodeFirst
{
    public class SampleContext : DbContext
    {
        public SampleContext() : base("MyShop")
        { }

        public DbSet<Customer> Customers { get; set; }
        public DbSet<Order> Orders { get; set; }
    }
}
```

Этот класс контекста представляет полный слой данных, который можно использовать в приложениях. Благодаря **DbContext**, можно запросить, изменить, удалить или вставить значения в базу данных. Обратите внимание на использование конструктора в этом классе с вызовом конструктора базового класса **DbContext** и передачей ему строкового параметра.

В этом параметре указывается либо имя базы данных, либо строка подключения к базе данных. В данном случае указывается явно имя базы данных, т.к. по умолчанию, при генерации базы данных EF использует имя приложения и контекста данных (например, CodeFirst.SampleContext), которое использовать неудобно.

3. Постройте приложение.

Установка подключения к базе данных

Для установки подключения обычно используется файл конфигурации приложения, в проектах для настольных приложений это файл *App.config*.

1. Откройте файл конфигурации *App.config* и после закрывающего тега `</configSections>` добавьте секцию `connectionStrings`, в которой с помощью элемента укажите подключение:

```
<connectionStrings>
   <add name="MyShop" connectionString="data
source=(localdb)\v11.0;Initial Catalog=userstore.mdf;Integrated
Security=True;"
providerName="System.Data.SqlClient"/>
 </connectionStrings>
```

В конструкторе класса контекста SampleContext была передана в качестве названия подключения строка "MyShop", поэтому данное название и указывается в атрибуте `name=" MyShop"`.

Работа с данными

На текущий момент есть классы данных и класс контекста, необходимые для работы с базой данных, которая еще не существует. Простое создание этих классов не приводит к автоматическому созданию (изменению) базы данных при запуске приложения. Чтобы создать базу данных, нужно добавить код работы с данными с помощью EF, например, написать код вставки данных в таблицу.

1. В поле класса формы объявите ссылки:
   a) на созданный ранее контекст данных:

```
SampleContext context;
```

   b) на массив байт, для загрузки фото заказчика:

```
byte[] Ph;
```

2. В обработчике события нажатия кнопки **Добавить данные** создайте экземпляр класса контекста базы данных, создайте объект – заказчика с передачей в его свойства соответствующих значений текстовых полей формы. После этого добавьте методом `Add()` созданного заказчика в базу данных и, наконец методом `SaveChanges()` сохраните изменения в базе данных:

```
try
{
        Customer customer = new Customer
        {
            Name = this.textBoxname.Text,
            Email = this.textBoxmail.Text,
            Age = Int32.Parse(this.textBoxage.Text),
```

```
                    Photo = Ph
                };
                context.Customers.Add(customer);
                context.SaveChanges();
                textBoxname.Text = String.Empty;
                textBoxmail.Text = String.Empty;
                textBoxage.Text = String.Empty;
            }
            catch(Exception ex)
            {
                MessageBox.Show("Ошибка: " + ex.ToString());
            }
        }
```

Данный код создает объект контекста **SampleContext** и использует его, чтобы добавить новые данные в таблицу **Customers**. Вызов метода `SaveChanges()` сохраняет изменения в базе данных, а при первой вставке данных вызов `SaveChanges()` создаст базу данных.

3. Добавьте обработчик нажатия кнопки **Выберите файл**, которая даст возможность загрузить изображение в базу данных.
4. В обработчике создайте объект стандартного диалогового окна для открытия файла, загрузите картинку в класс изображения Image и преобразуйте ее в массив байт:

```
OpenFileDialog diag = new OpenFileDialog();
if (diag.ShowDialog() == DialogResult.OK)
{
    Image bm = new Bitmap(diag.OpenFile());

    ImageConverter converter = new ImageConverter();
    Ph = (byte[])converter.ConvertTo(bm, typeof(byte[]));
}
```

5. Постройте и запустите приложение.
6. Введите данные о заказчике и нажмите кнопку **Добавить данные.**

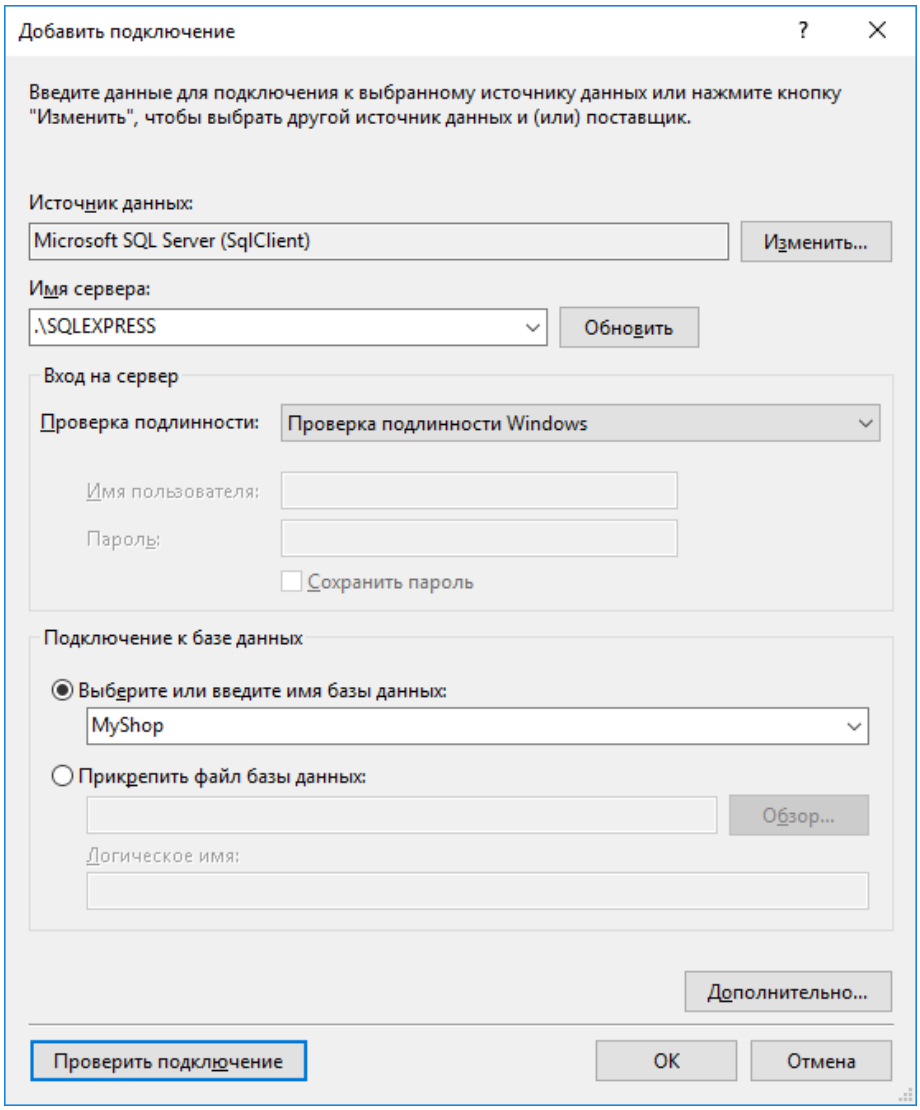
Обратите внимание, что при первом добавлении данных возникает заметная задержка, потому что в данном моменте и создается база данных.

7. Введите данные о втором заказчике. Обратите внимание, что данные добавились в этом случае быстро, т.к. они вставляются в уже существующую базу данных.

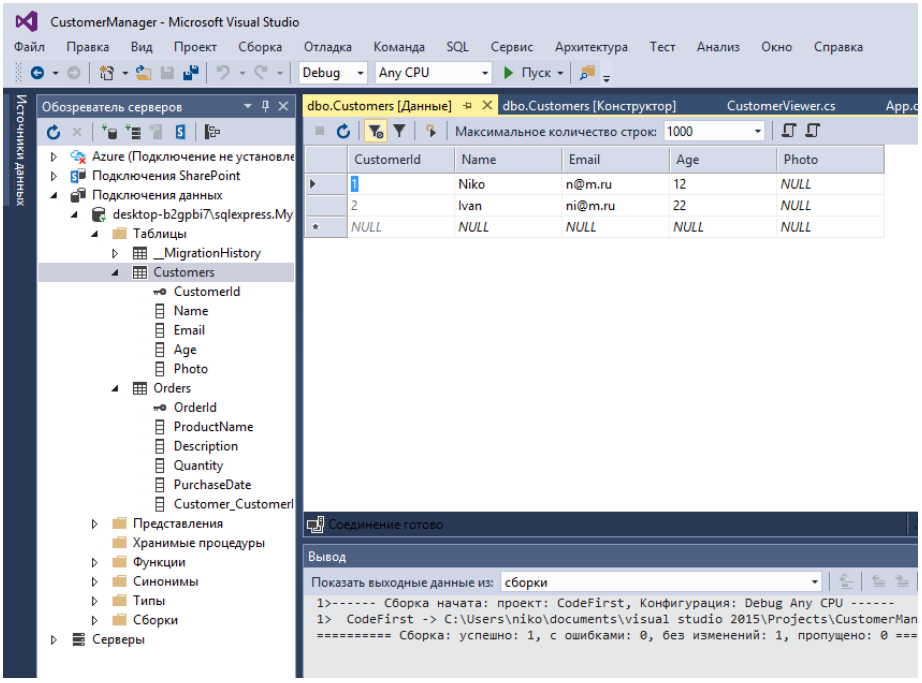
Соглашения о конфигурации сгенерированной базы данных

База данных создана. Далее вы просмотрите ее структуру.

1. Откройте окно Server Explorer в Visual Studio и нажмите на кнопке Connect to Database.
2. В открывшемся окне выберите поставщик SQL Server и настройте подключение к созданной базе данных MyShop, используя имя сервера ".\SQLEXPRESS" или (localdb)\v11.0 в окне **Add Connection**:



3. После этого в окне **Server Explorer** отобразится новое подключение:

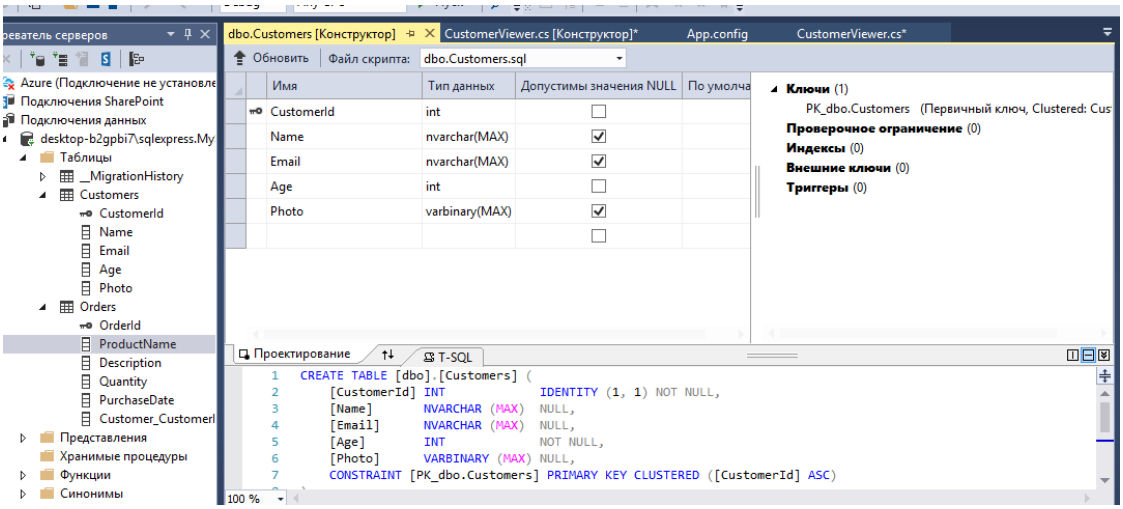


4. Проверьте, что в базе данных были созданы две таблицы, имеющие соответствующие столбцы.

Согласно соглашениям EF по именованию таблиц, их название генерируется в множественном числе (основываясь на правилах английского языка), т.е. для класса **Customer** генерируется таблица **Customers**, для класса **Order** генерируется таблица **Orders**.

Названия столбцов соответствуют названиям свойств сущностных классов.

5. В контекстном меню таблицы **Customers** выберете пункт *Open Table Defenition (Открыть определение таблицы)* и посмотрите, что типы данных .NET преобразованы в типы данных T-SQL: Int32 в INT, String в NVARCHAR(max), byte[] в VARBINARY(max) и т.д.:



6. Изучите структуру таблиц, обратите внимание на некоторые соглашения о проецировании сущностных классов, которые используются в Code-First.

Например, свойства `CustomerId` и `OrderId` в сущностном классе EF преобразовал в первичные ключи в базе данных (EF автоматически ищет подстроку “Id” в именах свойств модели с помощью механизма рефлексии). Эти поля используют автоинкремент с помощью инструкции IDENTITY (1,1) и не могут иметь значение NULL.

7. Откроете структуру таблицы **Orders** и проверьте, что EF создал внешний ключ `Customer_CustomerId`, ссылающий на таблицу **Customers**.

Это было достигнуто за счет того, что мы указали виртуальное свойство в классе **Customer** ссылающееся на класс **Order** и добавили обратное свойство в классе **Order**, ссылающееся на **Customer**. Благодаря тому, что тип виртуального свойства унаследован от **IEnumerable<T>**, EF догадался, что нужно реализовать отношение “один ко многим” (one-to-many) между этими таблицами. При подходе к именованию внешнего ключа EF использовал следующее соглашение:

```
[Имя навигационного свойства]_[Имя первичного ключа родительской таблицы]
```

Обратите внимание также на то, что EF сгенерировал еще одну таблицу с названием **__MigrationHistory**. Эта таблица хранит различные версии изменения структуры базы данных. В частности, в поле Model эта таблица хранит метаданные модели, представленные в виде двоичного объекта BLOB. Если позже вы измените модель в своем коде, EF вставит в эту таблицу новую запись, с метаданными новой модели.

8. С помощью команды контекстного меню *Показать таблицу данных* таблицы **Customer** просмотрите ее содержимое – вы увидите те данные, которые и были вами добавлены.

Отображение данных

Далее в этой части упражнения вы реализуете вывод данных из базы в подходящие для этого элементы управления.

1. В данном упражнении используется один элемент **GridView**, предназначенный для отображения данных из двух таблиц – **Customers** и **Orders**. Для того чтобы при выполнении программы у пользователя была возможность выводить попеременно данные из разных таблиц добавьте в класса формы метод, в котором при

проверке состояния элементов **RadioButton** менялся бы источник заполнения элемента **GridView**, например:

```
private void Output()
{
    if (this.CustomerradioButton.Checked == true)
        GridView.DataSource = context.Customers.ToList();
    else if (this.OrderradioButton.Checked == true)
        GridView.DataSource = context.Orders.ToList();
}
```

2. Добавьте обработчик нажатия кнопки **Показать данные** и в нем вызовите только что созданный метод `Output()`.
3. Также в этом же обработчике создайте LINQ-запрос на выборку заказчиков и сортировки их по имени:

```
var query = from b in context.Customers
            orderby b.FirstName
            select b;
```

4. Заполните элемент **ComboBox** (имя элемента `customerList`) результатом запроса:

```
customerList.DataSource = query.ToList();
```

5. Добавьте метод `Output()` в обработчик события нажатия кнопки **Добавить данные** после метода `SaveChanges()`.
6. Постройте и запустите приложение. Нажмите кнопку **Показать данные**. Информация о введенных ранее заказчиках должна отобразиться в реализованных элементах.
7. Введите информацию о новом заказчике в текстовые поля и нажмите кнопку **Добавить данные**. Информация о введенном заказчике должна отобразиться.

Реализация отношения «один ко многим»

В этой части упражнения вы реализуете связь заказчика с несколькими заказами. В учебном варианте для примера вы явно создадите заказы в коде приложения, в реальном проекте для ввода информации о заказе необходимо также как и для заказчика реализовать ввод данных с помощью элементов управления.

1. Добавьте обработчик события загрузки формы и в нем добавьте два заказа в базу данных, сохраните изменения и отобразите их в списке **ListBox** (`orderlistBox`):

```
private void CustomerViewer_Load(object sender, EventArgs e)
```

```
          {
        context.Orders.Add(new Order { ProductName = "Аудио", Quantity = 12,
PurchaseDate = DateTime.Parse("12.01.2016") });
        context.Orders.Add(new Order { ProductName = "Видео", Quantity = 22,
PurchaseDate = DateTime.Parse("10.01.2016") });
            context.SaveChanges();
           orderlistBox.DataSource = context.Orders.ToList();
          }
```

2. В код создания заказчика (в обработчике события нажатия кнопки **Добавить данные**) добавьте присвоение полю `Orders` выбранных заказов – элементов списка:

```
Orders = orderlistBox.SelectedItems.OfType<Order>().ToList()
```

3. Постройте и запустите приложение. Выберете переключатель **Orders**. Нажмите кнопку **Показать данные**. Информация о созданных заказах должны отобразиться в реализованных элементах.
4. Введите информацию о новом заказчике в текстовые поля, выделите заказы из списка и нажмите кнопку **Добавить данные**. Отобразите информацию о заказах или заказчиках в элементе **GridView**.

Изменение конфигурации базы данных

Реализованные элементы конфигурации базы данных можно переопределить с помощью средств Entity Framework: аннотации в виде атрибутов C# и строгая типизация с помощью Fluent API.

Использование аннотаций метаданных

Аннотации метаданных применяются непосредственно к классам и свойствам класса в виде атрибутов (атрибуты доступны в приложении при наличии ссылок на сборки System.ComponentModel.DataAnnotations.dll и EntityFramework.dll).

В этой части упражнения вы примените аннотации метаданных к классу **Customer** для того чтобы обеспечить следующие изменения в этом классе:

- Поле `Name` не должно иметь значение NULL в таблице базы данных, и длина поля не должна превышать 30 символов.
- Максимальная длина поля `email` не должна превышать 100 символов.
- Поле `Age` должно находиться в пределах от 8 до 100.
- Фотография должна хранится в типе изображения Image SQL Server, а не в VARBINARY (max).

1. Добавьте ссылки на пространства имен в файле Model.cs:

```
System.ComponentModel.DataAnnotations
System.ComponentModel.DataAnnotations.Schema
```

2. После этого измените структуру класса **Customer**, чтобы указать условия, приведенные выше:

```
[Required]
[MaxLength(30)]
public string Name { get; set; }

[MaxLength(100)]
public string Email { get; set; }

[Range(8, 100)]
public int Age { get; set; }

[Column(TypeName = "image")]
public byte[] Photo { get; set; }
```

3. Запустите приложение и попытаетесь вставить нового заказчика, вы получите исключение InvalidOperationException (проблема заключается в способе инициализации базы данных, используемом в Code-First):

The model backing the 'SampleContext' context has changed since the database was created. Consider using Code First Migrations to update the database.

Модель класса контекста SampleContext была изменена, когда база данных уже была создана. Рассмотрите возможность обновления базы данных с помощью Code First Migrations.

При генерации базы данных была добавлена таблица **__MigrationHistory**, в последней записи которой хранится структура модели созданной базы. После внесения изменений в класс **Customer** Entity Framework определил, что модель поменялась – она не соответствует последней записи в этой таблице и, следовательно, не может гарантировать, что ваша модель будет правильно отображаться в базе данных, в результате чего Entity Framework автоматически сгенерировал исключение.

Можно вручную удалить базу данных и запустить проект снова, при первой вставке данных сгенерируется новая база данных.

Другим решением проблемы, которое будет использовано в этом упражнении, является использование возможности Code-First с помощью статического метода `SetInitializer()` класса **Database** обнаруживать

изменения в модели данных и автоматически удалять, и генерировать новую базу данных.

По умолчанию, этому методу передается экземпляр класса **CreateDatabaseIfNotExists**, который указывает, что создавать базу данных нужно только в том случае, если базы данных не существует, а если модель изменилась, то нужно сгенерировать исключение.

4. В конструкторе формы главного приложения после создания контекста вызовите метод `SetInitializer()` класса **Database** и передайте ему экземпляр объекта **DropCreateDatabaseIfModelChanges**, который указывает Code-First, что при изменении модели данных, базу данных нужно будет удалить и воссоздать с новой структурой:

```
Database.SetInitializer(new
DropCreateDatabaseIfModelChanges<SampleContext>());
```

5. Постройте и запустите приложение. Заполните данные и вставьте их в базу. В этом случае Code First найдет разницу в новой модели и, с разрешения инициализатора, удалит и заново создаст базу данных.

Замечание. Если вы открыли таблицу базы данных для чтения данных где-то еще (например, в окне Server Explorer среды Visual Studio), Code First будет не в состоянии удалить базу данных, т.к. она используется другим процессом. В этом случае, возникнет заметная задержка при вставке данных, пока EF пытается удалить базу данных, а затем в конце концов будет сгенерировано следующее исключение:

SqlException: Cannot drop database "MyShop" because it is currently in use.

SqlException: Невозможно удалить базу данных "MyShop" т.к. она сейчас используется.

Редактирование модели – подход Code-Second

Далее в этом упражнении вы используете подход к работе, называемый *Code-Second,* когда вы вручную изменяете структуру базы данных с помощью T-SQL, а затем отражаете эти изменения в классах модели. Такой подход является единственной возможностью изменить структуру базы данных, не удаляя ее (чтобы не удалять уже вставленные данные).

Допустим, вам нужно изменить структуру таблицы **Customer** изменив имя столбца **Name** на **FirstName** и добавив новый столбец **LastName** (вспомните, что интерфейс формы изначально создавался для этого свойства).

Для решения этой задачи выполните следующие действия:

1. Используя программу SQL Server Management Studio или окно Server Explorer в Visual Studio удалите таблицу **__MigrationHistory**.
2. Выполните следующий SQL-код для изменения структуры таблиц (для этого щелкните правой кнопкой мыши по имени таблицы **Customer** в окне Server Explorer и выберите из контекстного меню команду **New Query**):

```
USE MyShop;
ALTER TABLE Customers ADD LastName NVARCHAR(30) NULL;
EXEC sp_rename @objname = 'Customers.Name', @newname = 'FirstName';
```

3. Внесите соответствующие изменения в класс **Customer:**

```
[Required]
[MaxLength(30)]
public string FirstName { get; set; }
public string LastName { get; set; }
```

4. Измените код заполнения базы данных:
   a) измените (выделены жирным шрифтом) код заполнения свойств модели:

```
Customer customer = new Customer
{
 FirstName = this.textBoxname.Text,
 LastName = this.textBoxlastname.Text,
 Email = this.textBoxmail.Text,
 Age = Int32.Parse(this.textBoxage.Text),
 Orders = orderlistBox.SelectedItems.OfType<Order>().ToList(),
 Photo = Ph
 };
```

   b) добавьте код для очищения поля фамилии:

```
textBoxlastname.Text = String.Empty;
```

   c) замените в коде идентификатор `Name` на `FirstName`,
   d) удалите вызов метода `SetInitializer()`.
5. Постройте и запустите приложение. Убедитесь, что вставка данных в базу работает корректно и при этом данные, которые вы вставили ранее сохранились.

Использование Fluent API

В этой части упражнения вы примените концепцию **Fluent API**, которая заключается в вызове ряда стандартных методов для описания настройки конфигурации базы данных. Основными из них являются методы `Entity()` и

`Property()`, первый дает выбрать объект для настройки, а второй – свойство выбранного объекта сущности.

Когда приходит время для построения модели, DbContext смотрит на структуру классов модели. Fluent API позволяет вмешаться в этот процесс и передать контексту дополнительные данные для конфигурации. Это становится возможным, благодаря переопределению метода `DbContext.OnModelCreating()`, который вызывается перед тем, как контекст построит сущностную модель данных. Этот метод является виртуальным, так что вы можете изменить его и вставить собственную логику, используя средства Fluent API.

Параметр типа **DbModelBuilder** в этом методе, позволяет добавлять настройки конфигурации.

Методы класса **DbModelBuilder** являются обобщенными и поддерживают указание лямбда-выражений в качестве параметров (принимают типы делегатов), благодаря чему обеспечивается быстрая настройка конфигурации.

1. В классе контекста **SampleContext** переопределите метод `OnModelCreating()`, в котором настройте свойство `LastName` сущности **Customer** аналогично тому как было настроено поле `FirstName` с помощью атрибутов:

```
        protected override void OnModelCreating(DbModelBuilder
modelBuilder)
        {
             modelBuilder.Entity<Customer>()
                .Property(c => c.LastName).IsRequired().HasMaxLength(30);
        }
```

2. Постройте приложение и протестируйте его работу.

Подходы с использованием аннотаций и Fluent API можно использовать в Code-First совместно, преимущество при указании одинаковых правил отдается Fluent API. Еще одним преимуществом Fluent API перед аннотациями является то, что он не засоряет код модели и организует взаимосвязь модели с контекстом данных.

Редактирование и удаление данных

В этой части упражнения вы реализуете возможность редактирования и удаления ранее добавленных данных.

Действия по редактированию и удалению начинаются с выделения записи данных в элементе **GridView**. Главной целью этой операции является

получение уникального идентификатора, который будет использоваться для поиска требуемой записи для редактирования и удаления.

1. Добавьте обработчик события выделения ячейки элемента **GridView.**
2. В обработчике присвойте переменной `customer` объект, находящийся в выделенной строке:

```
  private void orderGridView_CellClick(object sender,
DataGridViewCellEventArgs e)
    {
        if (GridView.CurrentRow == null) return;
        var customer = GridView.CurrentRow.DataBoundItem as Customer;
        if (customer == null) return;
```

3. Далее присвойте метке `labelid` уникальный идентификатор записи заказчика (`CustomerId`) и отобразите выделенного заказчика в соответствующих элементах:

```
        labelid.Text = Convert.ToString(customer.CustomerId);
        textBoxCustomer.Text = customer.ToString();

        textBoxname.Text = customer.FirstName;
        textBoxlastname.Text = customer.LastName;
        textBoxmail.Text = customer.Email;.
        textBoxage.Text = Convert.ToString(customer.Age);
    }
```

4. Добавьте обработчик нажатия кнопки **Редактировать**.
5. В обработчике присвойте переменной `id` значение элемента `labelid`, с помощью метода `Find(id)` найдите по значению `id` нужную запись заказчика и присвойте новые текущие значения элементов формы:

```
  private void buttonEdit_Click(object sender, EventArgs e)
  {
      if (labelid.Text == String.Empty) return;

      var id = Convert.ToInt32(labelid.Text);
      var customer = context.Customers.Find(id);
      if (customer == null) return;

      customer.FirstName = this.textBoxname.Text;
      customer.LastName = this.textBoxlastname.Text;
      customer.Email = this.textBoxmail.Text;
      customer.Age = Int32.Parse(this.textBoxage.Text);
```

6. Далее установите свойству состояния сущности `State` значение того, что сущность была изменена (`Modified`), сохраните изменения в базе данных и вызовите метод `Output()` для заполнения `GridView`:

```
        context.Entry(customer).State = EntityState.Modified;

        context.SaveChanges();
        Output();
    }
```

7. Добавьте обработчик нажатия кнопки **Удалить**.
8. В обработчике присвойте переменной `id` значение элемента `labelid`, с помощью метода `Find` найдите по значению `id` нужную запись заказчика, далее установите свойству состояния сущности `State` значение того, что сущность была помечена к удалению (`Deleted`), вызовите метод `SaveChanges()` для удаления сущности и вызовите метод `Output()` для заполнения **GridView**:

```
    private void buttonDel_Click(object sender, EventArgs e)
    {
        if (labelid.Text == String.Empty) return;

        var id = Convert.ToInt32(labelid.Text);
        var customer = context.Customers.Find(id);

        context.Entry(customer).State = EntityState.Deleted;
        context.SaveChanges();
        Output();
    }
```

9. Постройте приложение и протестируйте его работу. Отредактируйте информацию о заказчике, проверьте возможность удаления данных.

Упражнение 2. Использование наследования при создании модели

В этом упражнении вы добавите в модель новый класс, наследуемый от уже существующего класса и таким образом, измените структуру базы данных.

Генерация общей таблицы

В этой части упражнения на основе двух классов базового и производного будет сгенерирована одна таблица со специальным столбцом, позволяющим различить базовую и производную сущности.

1. Добавьте в файл модели новый класс, определяющий, например, важные заказы:

```
public class VipOrder: Order
{
    public string status { get; set; }
}
```

2. В класс контекста базы данных добавьте свойство с именем новой таблицы:

```
public DbSet<VipOrder> VipOrders { get; set; }
```

3. В обработчике события загрузки формы сразу после создания двух заказов создайте особый заказ и добавьте его в базу данных:

```
context.VipOrders.Add(new VipOrder { ProductName = "Авто", Quantity =
101, PurchaseDate = DateTime.Parse("10.01.2016"), status = "Высокий" });
```

4. Поскольку структура базы данных изменилась, то в конструкторе формы главного приложения после создания контекста вызовите метод `SetInitializer()` класса **Database** и передайте ему экземпляр объекта **DropCreateDatabaseIfModelChanges**, который указывает Code-First, что при изменении модели данных, базу данных нужно будет удалить и воссоздать с новой структурой:

```
Database.SetInitializer(new
DropCreateDatabaseIfModelChanges<SampleContext>());
```

5. Для возможности просмотра именно особых заказов добавьте на форму в группирующий элемент новый элемент выбора **RadioButton** (`ViporderradioButton`) и добавьте в метод `Output()` его проверку и изменение в зависимости от его значения элемента **GridView**:

```
else if (this.ViporderradioButton.Checked == true)
        GridView.DataSource = context.VipOrders.ToList();
```

6. Постройте приложение. Раскройте в окне **Server Explorer** построенную базу и проверьте, что созданы две таблицы **Customers** и **Orders** (для хранения и обычных и особых заказов используется одна таблица).
7. Откройте определение таблицы **Orders** и проверьте, что в нее добавлены поле `status` для особых заказов и поле `Discriminator` для отличия двух сущностей (маркер отличий).
8. Запустите приложение. Так как создается новая база данных, то заказчиков нет. Отобразите в сетке заказы с переключателем **Orders**, обратите внимание, что отобразятся все заказы, отобразите данные с

переключателем **VipOrders** – должны отобразится только особые заказы. Закройте приложение.

9. В окне **Server Explorer** откройте таблицу **Orders** в режиме отображения данных. Обратите внимание, что в столбце `Discriminator` указаны типы заказов, т.е. к какой сущности относится данная запись в таблице, а для типа **VipOrder** заполнено поле `status`.

Для каждого класса отдельная таблица

В этой части упражнения на основе двух классов базового и производного будет сгенерированы две таблицы – по одной на каждую сущность.

1. Для использования этого подхода перед классом **VipOrder** укажите атрибут Table с указанием имени таблицы `VipOrders`:

```
[Table("VipOrders")]
```

2. Постройте и запустите приложение. Функциональность приложения не изменилась. Закройте приложение.
3. Раскройте в окне **Server Explorer** построенную базу и проверьте, что созданы три таблицы **Customers, Orders** и **VipOrders** (для хранения и обычных и особых заказов используется своя таблица).
4. Проверьте структуру таблицы **Orders** и обратите внимание, что в этом случае нет поля `Discriminator`, а в таблице **VipOrders** указано поле `status`.

Упражнение 3. Построение EDM для работы с базой данных

В этом упражнении расматривается подход **Database first**, при котором на основе созданной ранее базы данных (база данных School) формируют ее модель – Entity Data Model (EDM). С помощью конструктора ADO.NET EDM Designer вы создадите концептуальную модель, модель хранения и сведения о сопоставлениях, которые собраны в файле EDMX.

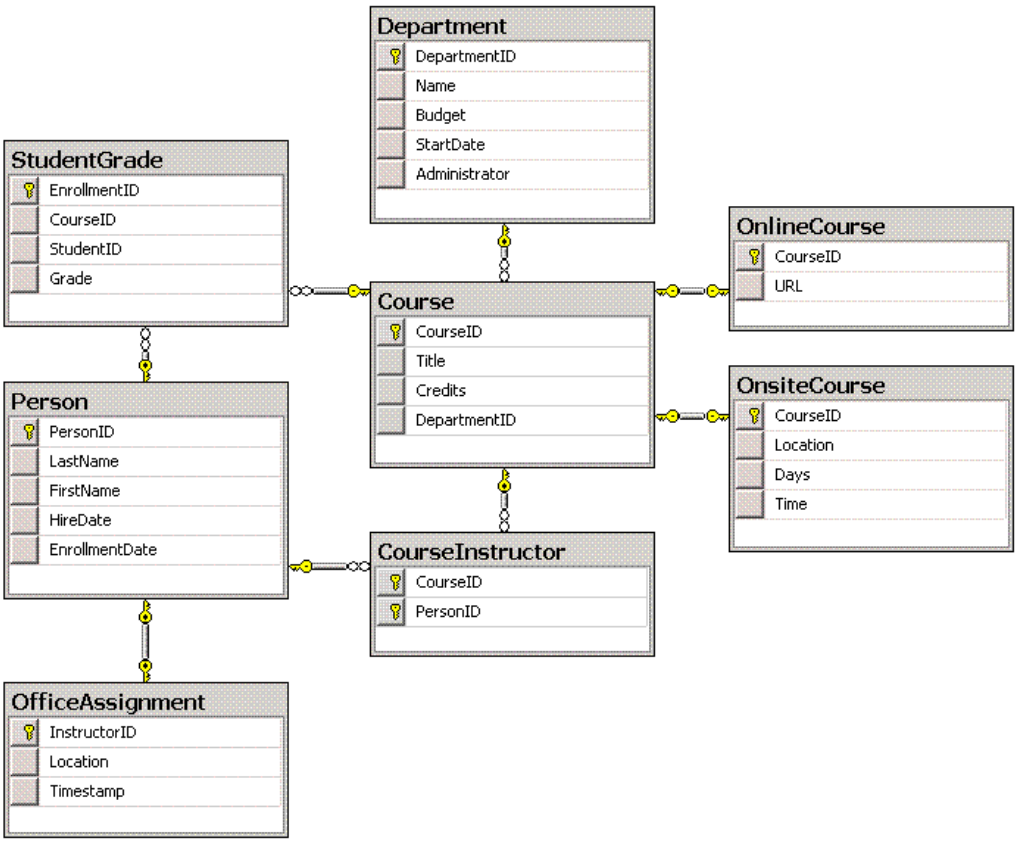
Вы разработаете новое приложение Windows Forms, в котором сформируеете запросы для доступа к данным в модели, выполните привязку результатов запросов к элементам управления для их отображения, а также реализуете возможность обновления в объектах и сохранение внесенных изменений в базу данных.

Построение базового приложения

В этой части упражнении создается схема базы данных **School** и загружаются данные в эту базу данных. Для этого используется среда SQL Server Management Studio, в которой выполняется скрипт Transact-SQL.

Полученная база данных School используется в качестве реляционного источника данных в остальной части упражнения.

1. Запустите среду SQL Server Management Studio и выполните скрипт Transact-SQL **School.sql** в папке Practices.
2. В окне **Обозреватель объектов** разверните узел только что подключенного экземпляра, разверните узел **Базы данных**, разверните узел **School**, разверните узел **Таблицы** и просмотрите список объектов таблиц в базе данных.
3. Изучите схему базы данных **School:**



4. Создайте новое WinForms-приложение (назовите его CourseManager).
5. Файлу формы присвойте имя **CourseViewer.cs**.
6. Настройте свойства формы: значение свойства **(Name)** – **CourseViewer,** значение свойства **Text** – **Course Viewer**.
7. Перенесите на форму элемент управления **ComboBox** на форму и измените имя элемента управления на **departmentList**.
8. Перенесите на форму кнопку, измените свойство (**Name**) этого элемента управления на **closeForm**, а значение свойства **Text** — на **Закрыть**.
9. Перенесите на форму элемент управления **DataGridView** на форму и измените имя элемента управления на **courseGridView**.
10. Постройте приложение.

Создание EDMX-файла базы данных

В этом упражнении используется мастер моделей EDM для формирования EDMX-файла, содержащего концептуальную модель, модель хранения и данные о сопоставлении. Файл определяет набор сопоставлений «один к одному» между сущностью и таблицей для концептуальной модели School и базой данных.

1. Выберите проект CourseManager в окне **Обозреватель решений**, щелкните его правой кнопкой мыши, укажите пункт **Добавить**, а затем выберите пункт **Создать элемент**.
2. Выберите в области **Шаблоны** пункт **Модель EDM ADO.NET**. В качестве имени модели введите **School.edmx** и нажмите кнопку **Добавить**. На экране откроется стартовая страница мастера моделей EDM.
3. В диалоговом окне **Выбор содержимого модели** выберите команду **Создать из базы данных**. Затем нажмите кнопку **Далее**.
4. Нажмите кнопку **Создать соединение**.
5. В диалоговом окне **Выбор источника данных** выберите источник данных и нажмите **Продолжить**.
6. В диалоговом окне **Свойства соединения** введите имя сервера, выберите метод проверки подлинности, введите имя базы данных **School** и нажмите кнопку **ОК**.
7. В диалоговом окне **Выбор подключения к данным** появятся заданные настройки подключения к базе данных.
8. Убедитесь, что установлен флажок **Сохранить настройки соединения сущности в App.config**, а значение установлено в **SchoolEntities**. Затем нажмите кнопку **Далее**. Откроется диалоговое окно **Выбор объектов базы данных**.
9. Выделите все таблицы и хранимые процедуры и убедитесь, что параметр **Пространство имен модели** имеет значение **SchoolModel**.
10. Установите флажки **Формировать имена объектов во множественном или единственном числе** и **Включить столбцы внешнего ключа в модель**.

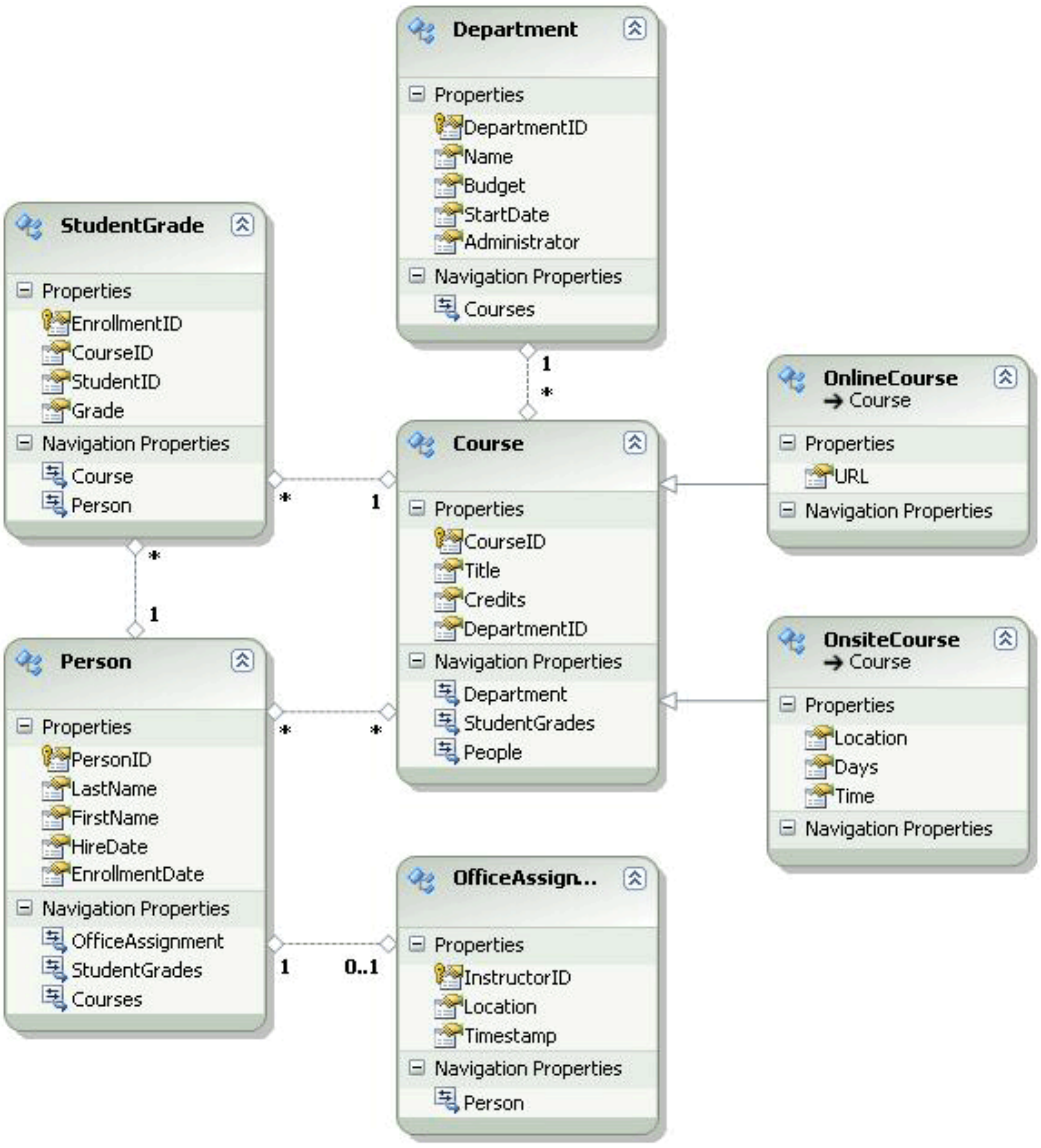
Мастер выполнил следующие действия:

- Добавляет ссылки на сборки **System.Data.Entity**, **System.Runtime.Serialization** и **System.Security**.
- Формирует файл School.edmx, который определяет концептуальную модель, модель хранения и их сопоставление.

- Создает файл кода уровня объекта, содержащий классы, сформированные на основе концептуальной модели. Файл с кодом уровня объекта можно просмотреть, развернув узел EDMX-файла в **обозревателе решений**.
- Создает файл App.Config.

11. Для просмотра EDMX-файла в конструкторе моделей EDM ADO.NET в **Обозревателе решений** дважды щелкните файл School.edmx.

Модель **School** откроется в окне конструктора моделей EDM ADO.NET, как на следующей схеме.



Содержимое хранения, сопоставления и концептуальное содержимое для модели **School** было успешно создано в проекте **CourseManager**.

Запросы к сущностям и ассоциациям

В этом упражнении создаются строго типизированные запросы к объектам среды CLR, которые представляют сущности и ассоциации в модели **School**, а элементы управления отображением привязываются к коллекциям объектов, возвращаемым этими запросами.

Будут созданы запросы, возвращающие объекты **Department** и **Course** и связывающие эти объекты с элементами управления.

Запрос по отделам в базе данных School

1. В файле с кодом для формы **CourseViewer** добавьте директиву **using** (C#), чтобы сослаться на модель, созданную из базы данных School, и пространство имен сущностей.

```
using System.Data.Objects;
using System.Data.Objects.DataClasses;
```

2. В начале определения класса для формы **CourseViewer** добавьте следующий код, создающий экземпляр **ObjectContext**:

```
private SchoolEntities schoolContext;
```

3. Добавьте обработчик события загрузки формы **courseViewer _Load** и реализуйте следующую функциональность:
    a. Инициализируйте экземпляр **ObjectContext**:

```
schoolContext = new SchoolEntities();
```

    b. Выполните запрос, возвращающий коллекцию отделов (упорядоченную по параметру `Name`):

```
var departmentQuery = from d in
schoolContext.Departments.Include("Courses")
                      orderby d.Name
                      select d;
```

    c. Инициализируйте представление списка в списке `departmentList` и привяжите коллекцию объектов **Department** к элементу управления `departmentList`:

```
try
{
    this.departmentList.DisplayMember = "Name";
    this.departmentList.DataSource =
((ObjectQuery)departmentQuery).Execute(MergeOption.AppendOnly);
}
catch (Exception ex)
{
```

```
        MessageBox.Show(ex.Message);
    }
```

Отображение курсов для выбранного отдела

4. В конструкторе формы **CourseViewer** дважды щелкните элемент управления **departmentList**. Будет создан метод обработчика события `departmentList_SelectedIndexChanged`.
5. Вставьте следующий код, загружающий курсы, связанные с выбранным отделом:

```
        try
        {
            Department department =
(Department)this.departmentList.SelectedItem;

            courseGridView.DataSource = department.Courses;

            courseGridView.Columns["Department"].Visible = false;
            courseGridView.Columns["StudentGrades"].Visible = false;
            courseGridView.Columns["OnlineCourse"].Visible = false;
            courseGridView.Columns["OnsiteCourse"].Visible = false;
            courseGridView.Columns["People"].Visible = false;
            courseGridView.Columns["DepartmentId"].Visible = false;

            courseGridView.AllowUserToDeleteRows = false;
courseGridView.AutoResizeColumns(DataGridViewAutoSizeColumnsMode.AllCells);
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
```

Вставка и обновление данных

В этом упражнении в базе данных будут сохранены изменения, внесенные в объекты **Course**, привязанные к элементу управления **DataGridView**.

1. Перенесите кнопку на форму, измените свойство (**Name**) этого элемента управления на **saveChanges**, а значение свойства **Text** — на *Обновить*.
2. Добавьте обработчик события `saveChanges_Click` и вставьте код, сохраняющий изменения объекта в базе данных:

```
try
{
    schoolContext.SaveChanges();
    MessageBox.Show("Changes saved to the database.");
    this.Refresh();
}
catch(Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

3. Добавьте обработчик события `closeForm_Click` и введите код для удаления контента объекта и закрытия формы:

```
this.Close();
schoolContext.Dispose();
```

4. Постройте и протестируйте работу приложения.
5. После загрузки формы выберите отдел в элементе управления **ComboBox,** просмотрите сетку, должны отображаться курсы, принадлежащие данному отделу.
6. В сетке добавьте новый курс и нажмите кнопку **Обновить.** Убедитесь, что отображается окно сообщения с уведомлением о сохранении изменений. Повторно запустите приложение и убедитесь, что изменения сохраняются в базе данных.

Литература

1. Шилд Г. C# 4.0 Полное руководство. – М.: Вильямс, 2011. – 1056с.
2. Малик С. Microsoft ADO.NET 2.0 для профессионалов.: пер. с англ. — М.: Вильямс, 2006 – 560 с.
3. Уотсон К. и др. Visual C# 2010. Полный курс. – М.: Диалектика, 2010. – 960с.
4. Троелсен Э. Язык программирования C# 2010 и платформа .NET 4.0 Framework. – М.: Вильямс, 2010. – 1392 с.