**Лабораторная работа № 4. Работа со строками таблиц**

**Цель работы:**

– получить практические навыки манипулирования строками таблиц объекта DataSet.

**Теоретические сведения**

1. **Добавление записей в таблицы**

Для добавления записи (строки) в таблицу необходимо сначала создать пустую строку.

***DataRow row =empTable.NewRow();***

***DataRow deptRow=deptsTable.NewRow();***

Затем нужно заполнить поля строки данными

***//Заполнение строк***

***row["LName"] = "Мороз";***

***row["FName"] = "Василий";***

***row["MName"] = "Петрович";***

***row["BirthDate"] = "30.12.1956";***

***row["DeptID"] = 1;***

***deptRow["ID"] = 1;***

***deptRow["Dept"] = "Дирекция";***

Затем нужно добавить строки в таблицу

***//Добавление строк в таблицы***

***dsShop.Tables[0].Rows.Add(row);***

***dsShop.Tables[1].Rows.Add(deptRow);***

Форма с отображением созданных таблиц может иметь вид, показанный на рисунке 4.1

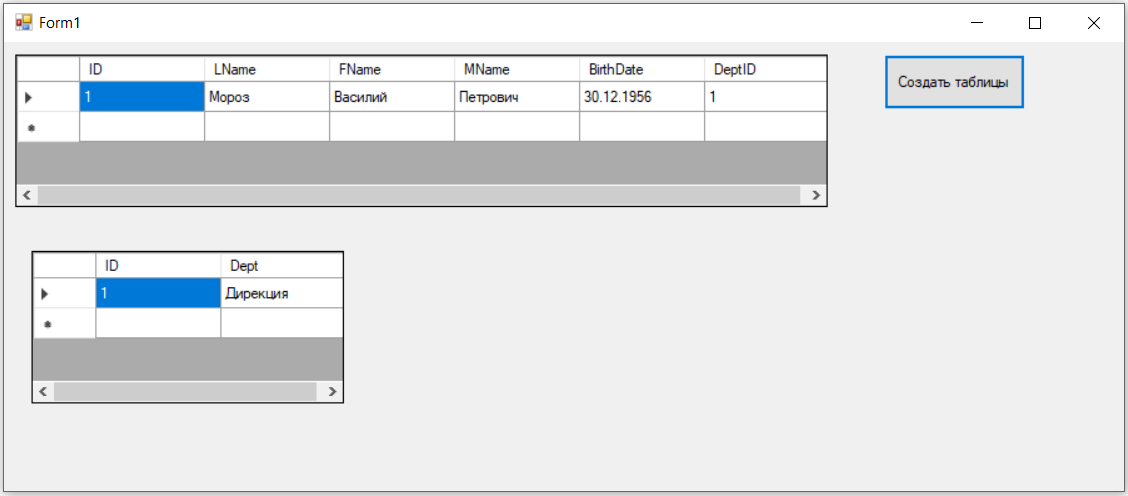


Рисунок 4.1. Форма с отображением созданных таблиц.

Полный код этой формы представлен ниже.

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace MyShop

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

//Создание DataSet

DataSet dsShop = new DataSet("Shop");

private void btCreateTabs\_Click(object sender, EventArgs e)

{

//Создание таблицы Employees

DataTable empTable = dsShop.Tables.Add("Employees");

DataColumn empID = empTable.Columns.Add("EmployeeID", typeof(Int32));

empID.AutoIncrement = true;

empID.AutoIncrementSeed = 200;

empID.AutoIncrementStep = 3;

empTable.Columns.Add("LName", typeof(string));

empTable.Columns.Add("FName", typeof(string));

empTable.Columns.Add("MName", typeof(string));

empTable.Columns.Add("BirthDate", typeof(DateTime));

empTable.Columns.Add("DeptID", typeof(Int32));

empTable.PrimaryKey = new DataColumn[] { empID };

//Создание таблицы Depts

DataTable deptsTable = dsShop.Tables.Add("Depts");

DataColumn deptID = deptsTable.Columns.Add("ID", typeof(Int32));

deptID.AutoIncrement = true;

deptID.AutoIncrementSeed = 200;

deptID.AutoIncrementStep = 3;

deptsTable.Columns.Add("Dept", typeof(string));

deptsTable.PrimaryKey = new DataColumn[] { deptID };

//Создание внешнего ключа

ForeignKeyConstraint DeptEmpFK = new ForeignKeyConstraint("DeptEmpFK",

dsShop.Tables["Depts"].Columns["ID"],

dsShop.Tables["Employees"].Columns["DeptID"]);

DeptEmpFK.DeleteRule = Rule.None;

dsShop.Tables["Employees"].Constraints.Add(DeptEmpFK);

//Создание пустых строк

DataRow row = dsShop.Tables[0].NewRow();

DataRow deptRow = dsShop.Tables[1].NewRow();

//Заполнение строк

row["LName"] = "Мороз";

row["FName"] = "Василий";

row["MName"] = "Петрович";

row["BirthDate"] = "30.12.1956";

row["DeptID"] = 1;

deptRow["ID"] = 1;

deptRow["Dept"] = "Дирекция";

//Добавление строк в таблицы

dsShop.Tables["Depts"].Rows.Add(deptRow);

dsShop.Tables["Employees"].Rows.Add(row);

//Отображение таблиц

dataGridView1.DataSource = dsShop.Tables["Employees"];

dataGridView2.DataSource = dsShop.Tables["Depts"];

}

}

}

# Состояния и версии строк

ADO.NET управляет строками таблиц с помощью состояний и версий строк. Состояние строки указывает на статус строки, а версии строк хранят значения изменения строки: исходное(Original), текущее (Current), применяемое по умолчанию(Default) и предлагаемое (Proposed) значения. Например, после внесения изменения в столбец строки эта строка будет иметь состояние Modified и две версии: Current, содержащую текущие значения, и Original, содержащую значения этой строки до изменения столбца.

Каждый объект [DataRow](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarow) имеет свойство [RowState](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarow.rowstate), которое отображает текущее состояние строки. В следующей таблице кратко описано каждое из значений перечисления RowState.

| **Значение свойства RowState** | **Описание** |
| --- | --- |
| Unchanged | Не было выполнено никаких изменений с момента последнего вызова метода AcceptChanges или с момента создания строки методом DataAdapter.Fill. |
| Added | Строка добавлена к таблице, но метод AcceptChanges не вызывался. |
| Modified | Изменены некоторые элементы строки. |
| Deleted | Строка удалена из таблицы, но метод AcceptChanges не вызывался. |
| Detached | Строка не принадлежит ни к одной коллекции DataRowCollection. Свойство RowState только что созданной строки имеет значение Detached. После добавления новой строки DataRow к коллекции DataRowCollection с помощью вызова метода Add свойству RowState присваивается значение Added. Значение Detached также присваивается строке, удаленной из DataRowCollection с помощью метода Remove или путем последовательного вызова методов Delete и AcceptChanges. |

В следующей таблице кратко описано каждое из значений перечисления DataRowVersion.

| **Значение DataRowVersion** | **Описание** |
| --- | --- |
| Current | Текущие значения строки. Эта версия не существует для строк, у которых RowState равно Deleted. |
| Default | Версия по умолчанию для конкретной строки. Версия по умолчанию для строки Added, Modified и Deleted представляет собой Current. Версия по умолчанию для строки Detached - Proposed. |
| Original | Исходные значения строки. Эта версия не существует для строк, у которых RowState равно Added. |
| Proposed | Предложенные значения строки. Эта версия строки существует в течение операции изменения строки или для строки, не содержащейся в коллекции DataRowCollection. |

Можно просматривать разные версии строки, передавая параметр [DataRowVersion](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarowversion) со ссылкой на столбец, как показано в следующем примере.

***DataRow emptRow = empTable.Rows[0];***

***string empID = empRow["EmployeeID", DataRowVersion.Original].ToString();***

Можно проверить, имеет ли DataRow конкретную версию строки, вызвав метод [HasVersion](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarow.hasversion) и передав ему DataRowVersion в качестве аргумента. Например,

***empRow.HasVersion(empRowVersion.Original)***

Если такая версия строк имеетс, то метод HasVersion возвращает true, иначе – false.

1. **Изменение и удаление записей**

При выполнении изменений значений столбцов в [DataRow](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarow), изменения немедленно помещаются в текущую (Current) версию строки. Затем для [DataRowState](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarowstate) задается значение Modified, и изменения принимаются или отклоняются с помощью методов [AcceptChanges](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarow.acceptchanges) или [RejectChanges](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarow.rejectchanges) объекта DataRow.

**DataRow** также предоставляет три метода, которые можно использовать для управления версиями строки при ее редактировании. Такими методами являются: [BeginEdit](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarow.beginedit), [EndEdit](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarow.endedit) и [CancelEdit](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarow.canceledit)

При непосредственном изменении значений столбцов в **DataRow** объект **DataRow** управляет версиями строк **Current**, **Default**, и **Original**. В дополнение к этим версиям строк методы **BeginEdit**, **EndEdit** и **CancelEdit** используют четвертую версию строки: **Proposed**.

**Proposed** версия строки существует во время операции изменения, которая начинается с вызова **BeginEdit** и заканчивается либо с помощью **EndEdit, либо CancelEdit,** либо путем вызова **метода AcceptChanges** или **RejectChanges**.

|  |  |
| --- | --- |
| Метод | Действие |
| BeginEdit | Создается версия строки Proposed |
| CancelEdit | Версия строки Proposed удаляется |
| EndEdit | Версия строки Proposed становится версией Current |
| AcceptChanges | Неявно вызывается метод EndEdit. Производятся все изменения, произведенные после последнего вызова [AcceptChanges()](https://docs.microsoft.com/en-us/dotnet/api/system.data.datatable.acceptchanges?view=net-5.0#System_Data_DataTable_AcceptChanges).Строки с состояниями  Added и Modified получают состояние Unchanged, строки с состоянием Deleted удаляются. |
| RejectChanges | Неявно вызывается CancelEdit. Отменяются все изменения, сделанные с момента загрузки таблицы или последнего вызова [AcceptChanges()](https://docs.microsoft.com/en-us/dotnet/api/system.data.datatable.acceptchanges?view=net-5.0#System_Data_DataTable_AcceptChanges).  Новые строки удаляются. Строки с состоянием Modified или Deleted возвращаются в исходное состояние. |

Во время операции редактирования можно применить логику проверки к отдельным столбцам, оценивая **proposedValue в** событии **ColumnChanged** объекта **DataTable**.

Можно подтвердить изменения, вызвав **EndEdit**, или отменить их, вызвав **CancelEdit**. Обратите внимание, что хотя **EndEdit** подтверждает изменения, **DataSet** фактически не принимает изменения, пока не будет вызван метод **AcceptChanges**. Обратите внимание, что если метод **AcceptChanges** вызывается до завершения операции Edit с **EndEdit** или **CancelEdit**, то редактирование завершается и **Proposed** значения строки принимаются как для **Current** , так и для **Original** версий строк. Аналогичным образом, вызов **RejectChanges** завершает изменение и отменяет **Current** и **Orignal** версию строки. Вызов метода **EndEdit** или **CancelEdit** после вызова **метода AcceptChanges** или **RejectChanges** не действует, так как редактирование уже завершено.

В следующем примере показано, как использовать **BeginEdit** с **EndEdit** и **CancelEdit**. В примере также проверяется **ProposedValue** в событии **ColumnChanged** и принимается решение, следует ли отменить изменение.

Если метод AcceptChanges вызывается для [DataSet](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.dataset), [DataTable](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datatable) или [DataRow](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarow), удаляются все строки со значением состояния Deleted. Оставшимся строкам присваивается состояние Unchanged, а значения в версии строки Original перезаписываются значениями версии строки Current. Если вызывается метод RejectChanges, удаляются все строки со значением состояния Added. Оставшимся строкам присваивается состояние Unchanged, а значения в версии строки Current перезаписываются значениями версии строки Original.

***Удаление строк***

Существует два метода, которые [DataRow](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarow) можно использовать для удаления объекта из объекта [DataTable](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datatable): метод **Remove** объекта [DataRowCollection](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarowcollection) и метод [Delete](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarow.delete) объекта **DataRow** . В то время как метод [Remove](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarowcollection.remove) удаляет **DataRow** из [DataRowCollection](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarowcollection), метод [Delete](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarow.delete) помечает только строку для удаления. Фактическое удаление происходит, когда приложение вызывает метод **AcceptChanges**.

Использование метода [Delete](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarow.delete) позволяет программно проверять, какие строки помечены для удаления, перед их фактическим удалением. Когда строка отмечена для удаления, ее свойство [RowState](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarow.rowstate) имеет значение [Delete](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarow.delete)d.

В следующем примере показано, как вызвать метод **Delete** для **DataRow** , чтобы изменить его свойство **RowState** на **deleted**.

***empRow.Delete();***

Метод **Remove** объекта [DataRowCollection](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.datarowcollection) принимает **DataRow** в качестве аргумента и удаляет его из коллекции, как показано в следующем примере.

***empTable.Rows.Remove(empRow);***

Если строка помечена для удаления и вызывается метод **AcceptChanges** объекта **DataTable** , то строка удаляется из **DataTable**. В отличие от этого, при вызове RejectChanges **RowState** строки восстанавливается до того, как она была помечена как **Deleted**.

Для работы сос троками можно использовать их по индексы:

|  |  |
| --- | --- |
|  | ***DataRow row = Employees.Rows[0]; // первая строка*** |

Получив строку по индексу, можно изменить ее ячейки:

|  |
| --- |
| ***Employees.Rows[0][4] = 2;*** //пятой ячейке первой строки присваивается значение 2 |

И также можно удалять строку:

***Employees.Rows.RemoveAt(1);*** // удаление второй строки по индексу

***// другой сопосб удаления***

***DataRow row = empTable.Rows[0];***

***empTable.Rows.Remove(row);***

1. **Метод SELECT**

Класс DataTable предлагает удобный метод Select(), позволяющий искать строки на основе SQL-выражения. Выражение, которое используется с методом Select(), играет ту же роль, что и конструкция WHERE в операторе SELECT, но имеет дело с данными в памяти, находящимися в объекте DataTable (так что никаких операций с базой данных не производится).

Используя метод Select() объекта DataTable можно найти строки, которые соответствуют определенному критерию. Например, можно получить строки, в которых идентификатор отдела (DeptID) равен 2. Для отображения этих строк можно создать таблицу, заполнить ее и затем отобразить в элементе dataGridView

DataTable selempTable = dsShop.Tables.Add("SelEmp");

DataRow[] selEmp = dsShop.Tables["Employees"].Select("DeptID=1");

DataColumn selempID = selempTable.Columns.Add("EmployeeID", typeof(Int32));

selempID.AutoIncrement = true;

selempID.AutoIncrementSeed = 200;

selempID.AutoIncrementStep = 3;

selempTable.Columns.Add("LName", typeof(string));

selempTable.Columns.Add("FName", typeof(string));

selempTable.Columns.Add("MName", typeof(string));

selempTable.Columns.Add("BirthDate", typeof(DateTime));

selempTable.Columns.Add("DeptID", typeof(Int32));

selempTable.PrimaryKey = new DataColumn[] { selempID };

foreach (DataRow row in selEmp)

{

DataRow row2 = dsShop.Tables["SelEmp"].NewRow();

row2[0] = row[0];

row2[1] = row[1];

row2[2] = row[2];

row2[3] = row[3];

row2[4] = row[4];

row2[5] = row[5];

dsShop.Tables["SelEmp"].Rows.Add(row2);

}

dataGridView3.DataSource = dsShop.Tables["SelEmp"];

1. **Обновление источника данных**

Получив данные в DataSet, можно производить с ними различными операции: удалять, изменять, добавлять новые записи. Однако все сделанные изменения автоматически не будут сохраняться в БД. Для сохранения данных нужно еще вызвать метод **Update** объекта SqlDataAdapter, который заполнял DataSet.

Для модификации данных в БД в соответствии с изменениями в DataSet SqlDataAdapter использует команды InsertCommand, UpdateCommand и DeleteCommand. Можно вручную определить для этих команд sql-выражения, либо воспользоваться классом **SqlCommandBuilder**, который позволяет автоматически сгенерировать нужные выражения.

Пример использования SqlCommandBuilder:

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

static public string connectionString = "Data Source=MyPC\\SQLEXPRESS;Initial Catalog=shop;Integrated Security=True";

static SqlConnection con = new SqlConnection(connectionString);

static string command = "SELECT \* FROM Employees";

static SqlDataAdapter adapter = new SqlDataAdapter(command, con);

public DataSet ds = new DataSet();

private void btRead\_Click(object sender, EventArgs e)

{

// Заполнение DataSet

adapter.Fill(ds, "Employees");

dataGridView1.DataSource = ds.Tables["Employees"];

}

private void btAdd\_Click(object sender, EventArgs e)

{

SqlCommandBuilder builder = new SqlCommandBuilder(adapter);

DataTable EmployeeTable = ds.Tables["Employees"];

DataRow row = EmployeeTable.NewRow();

row["FName"] = "Дмитрий";

row["LName"] = "Петров";

EmployeeTable.Rows.Add(row);

adapter.Update(ds, "Employees");

}

private void btClear\_Click(object sender, EventArgs e)

{

ds.Clear();

}

}

В этом примере при нажатии кнопки btAdd после загрузки данных создается объект SqiCommandBuilder, затем создается новая строка, которая затем добавляется в DataTable. При вызове у адаптера метода Update() происходит анализ изменений, которые произошли. И после этого выполняется соответствующая команда. В данном случае так как идет добавление новой строки, то будет выполняться команда InsertCommand. Однако в данном коде мы нигде явным образом команду не задается, ее автоматически формирует SqlCommandBuilder. Для применения этого класса достаточно вызвать его конструктор, в который передается нужный адаптер:

1. **Работа со связаными таблицами**

DataSet может содержать множество таблиц, которые могут быть связаны различными отношениями.

Рассмотрим пример, в котором показано применение DataSet для работы со связанными таблицами. В примере показано, как извлекать некоторые записи из таблиц groops и goods базы данных shop, а также как создавать отношения между ними для организации простой навигации от записи о категории к ее дочерним записям о товарах, чтобы создать простой отчет:

Структура и содержимое таблиц groups и goods показаны ниже

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

***private void btRead\_Click(object sender, EventArgs e)***

***{ string connectionString = "Data Source=MYPC\\SQLEXPRESS;Initial Catalog=Shop;Integrated Security=True";***

***SqlConnection con = new SqlConnection(connectionString);***

***string sqlGroup = "SELECT groupID, goodsgroup FROM groups"; string sqlGood = "SELECT good, groupId FROM goods";***

***SqlDataAdapter adapter = new SqlDataAdapter(sqlGroup, con);***

***DataSet dataset = new DataSet();***

***con.Open();***

***adapter.Fill(dataset, "groups"); // Наполнение DataSet данными из таблицы groups***

***adapter.SelectCommand.CommandText = sqlGood; // Получение данных таблицы goods.***

***adapter.Fill(dataset, "goods");***

***// Определение связи между таблицами***

***DataRelation relat = new DataRelation("GroupGoods",***

***dataset.Tables["groups"].Columns["groupID"],***

***dataset.Tables["goods"].Columns["groupID"]);***

***dataset.Relations.Add(relat); // Добавление отношения в DataSet***

***foreach (DataRow row in dataset.Tables["groups"].Rows)***

***{***

***listBox1.Items.Add(row["goodsgroup"]);***

***DataRow[] childRows = row.GetChildRows(relat); // Получение связанных данных***

***foreach (DataRow childRow in childRows) listBox1.Items.Add(" " + childRow["good"]);***

***}***

***con.Close();***

***}***

На первом шаге осуществляется инициализация объектов ADO.NET и объявление двух SQL-запросов (для извлечения групп и товаров). Далее код выполняет оба запроса, добавляя две таблицы к DataSet. Соединение открывается явно вначале и закрывается после двух операций, обеспечивая тем самым наилучшую производительность.

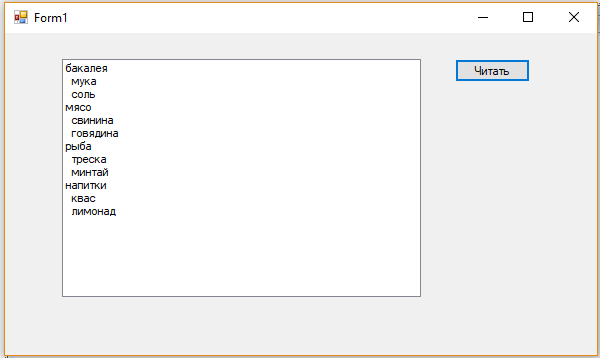
В этом примере один и тот же объект DataAdapter применяется для наполнения обеих таблиц. Это совершенно законный прием, к тому же он имеет смысл в настоящем сценарии, поскольку не нужно повторно использовать DataAdapter для обновления источника данных. Однако если бы DataAdapter применялся как для извлечения данных, так и для отправки изменений, то было бы необходимо для каждой из таблиц предусмотреть отдельный DataAdapter, что позволило бы выполнять соответствующие команды вставки, обновления и удаления для каждой таблицы.

В коде мы имеем DataSet с двумя таблицами. В базе данных Shop эти две таблицы связаны отношением по полю groupid. Оно является первичным ключом таблицы groups и внешним ключом таблицы goods. К сожалению, в ADO.NET не предусмотрено никакого способа прочитать информацию об отношениях между таблицами в источнике данных, чтобы применить ее к DataSet автоматически. Поэтому для представления этого отношения приходится вручную создавать DataRelation.

Отношение создается путем определения объекта DataRelation и добавления его к коллекции DataSet.Relations. При создании DataRelation конструктору передаются три аргумента: имя отношения, DataColumn первичного ключа родительской таблицы и DataColumn внешнего ключа дочерней таблицы.

Внутри блока можно обратиться к связанным записям о товарах для текущей категории, вызвав метод DataRow.GetChildRows(). Этот метод ищет данные — соответствующие строки в памяти, в связанном объекте DataTable. Получив массив записей о товарах, можно пройтись по ним с помощью вложенного цикла foreach. Это намного проще, чем код, который понадобился бы для поиска этой информации в отдельном объекте или для выполнения множества запросов с традиционным доступом через соединение.

Внешний вид формы показан ниже



Вопрос, который часто задают программисты-новички в ADO.NET звучит так: когда нужно использовать запросы c использованием JOIN, а когда — объекты Data Relation? Наиболее важное условие для правильного ответа — собираетесь ли вы обновлять извлеченные данные? Если да, то применение отдельных таблиц и объекта DataRelation всегда обеспечивает большую гибкость. Если нет, то можно применять любой подход, хотя запрос с использованием JOIN и может оказаться более эффективным, потому что включает лишь одно обращение к базе по сети, в то время как вариант с DataRelation требует два, чтобы наполнить отдельные таблицы.

Когда вы добавляете отношение между таблицами в DataSet, то при этом ограничены правилами ссылочной целостности. Например, вы не можете удалить родительскую запись, если существуют связанные с ней дочерние строки, и не можете создать дочернюю запись, ссылающуюся на несуществующую родительскую. Это может стать причиной проблем, если DataSet содержит лишь частичные данные.

Например, если имеется полный список клиентских заказов, но только часть списка клиентов, может случиться так, что заказ будет ссылаться на заказчика, которого не существует, потому что запись о нем в DataSet отсутствует. Один из способов обойти эту проблему — создать DataRelation без соответствующих ограничений. Чтобы сделать это, используйте конструктор DataRelation, который принимает булевский параметр createConstraints, и установите его значение в false, как показано ниже:

***DataRelation relat = new DataRelation("CatProds",***

***dataset.Tables["Categories"].Columns["CategoryID"],***

***dataset.Tables["Products"].Columns["CategoryID"], false);***

Другой подход предусматривает отключение всякого рода проверок ограничений (в том числе проверки уникальных значений) за счет установки свойства DataSet.EnableConstraints в false перед добавлением отношения.

**Задание на лабораторную работу**

1. Разработать форму (WindowsForms) для отображения данных одной из таблиц базы данных (из индивидуального задания), содержащей внешний ключ.

2. Создать в форме объект DataSet и в нем два объекта DataTable, структуры которых соответствуют двум связанным таблицам индивидуального задания и загрузить в них данные из базы данных.

3. Разместить в форме текстовые поля и кнопки для выполнения операций создания, вставки, отмены вставки, изменения и отмены изменения, удаления и отмены удаления записей из таблиц, а также элементы dataGridView для отображения данных таблиц в DataSet;

4. Создать отображения данных связанных таблиц в элементe listBox таким образом, чтобы под каждой родительской записью отображались все дочерние записи.

5. Создать элементы управления для обновления базы данных.

**Содержание отчета**

1. Тема и цель работы
2. Задание на лабораторную работу
3. Содержимое таблицы базы данных.
4. Внешний вид формы и программный код для каждого задания.
5. Выводы

**Контрольные вопросы**

1. Каков порядок добавления записей в объект DataTable?
2. Какие существуют состояния строк и что они означают?
3. Какие существуют версии строк и что они означают?
4. Какие методы применяются для изменения и удаления записей?
5. Как и для чего используется метод Select объекта DataTable?
6. Какой метод используется для записи данных из объекта DataSet в базу данных?