Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информатики

А. А. Волосевич, С. В. Актанорович

СРЕДСТВА ПЛАТФОРМЫ .NET ДЛЯ РАБОТЫ С БАЗАМИ ДАННЫХ

Методическое пособие
по дисциплинам «Инструменты и средства программирования»
и «Избранные главы информатики»
для студентов специальности 1-31 03 04 «Информатика»
всех форм обучения

Минск БГУИР 2011

УДК 004.45(076)
ББК 32.973.26-018.2я73
В68

Р е ц е н з е н т:
заведующий кафедрой многопроцессорных систем и сетей
Белорусского государственного университета,
кандидат технических наук Л. Ф. Зимянин

Волосевич, А. А.

В68 Средства платформы .NET для работы с базами данных : метод. пособие по дисц. «Инструменты и средства программирования» и «Избранные главы информатики» для студ. спец. 1-31 03 04 «Информатика» всех форм обуч. / А. А. Волосевич, С. В. Актанорович. – Минск : БГУИР, 2011. – 52 с. : ил.

ISBN 978-985-488-652-7.

Рассматривается технология ADO.NET, которая предназначена для работы с реляционными базами и другими сходными источниками данных.

ADO.NET – часть платформы Microsoft .NET Framework. Содержится справочная информация об основных компонентах и классах ADO.NET, приводятся примеры использования данной технологии.

Может быть рекомендовано студентам, магистрантам и преподавателям технических специальностей, а также разработчикам приложений, использующих технологию ADO.NET для доступа к данным.

УДК 004.45(076)
ББК 32.973.26-018.2я73

ISBN 978-985-488-652-7

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Практически любое современное бизнес-приложение использует при функционировании некий источник данных. Очень часто в качестве такого источника выступает реляционная база данных. Платформа .NET Framework – это средство для разработки бизнес-приложений, созданное компанией Microsoft. Важной частью .NET Framework является технология ADO.NET, применяемая для работы с базами данных.

В данном пособии проанализированы основные компоненты технологии ADO.NET, включая архитектуру, классы поставщиков данных и элементы рассоединённого набора данных. Подробное теоретическое описание компонентов сопровождается примерами кода на языке C#, представляющими практические пошаговые инструкции их использования. Материал, изложенный в пособии, позволяет создавать слой доступа к данным в бизнес-приложениях начального и среднего уровня, разработанных с использованием Microsoft .NET Framework.

Во многих примерах пособия используется учебная база данных `Library`, которая хранит информацию о книгах, категориях книг и авторах. Схема базы `Library` приведена на рис. В.1.

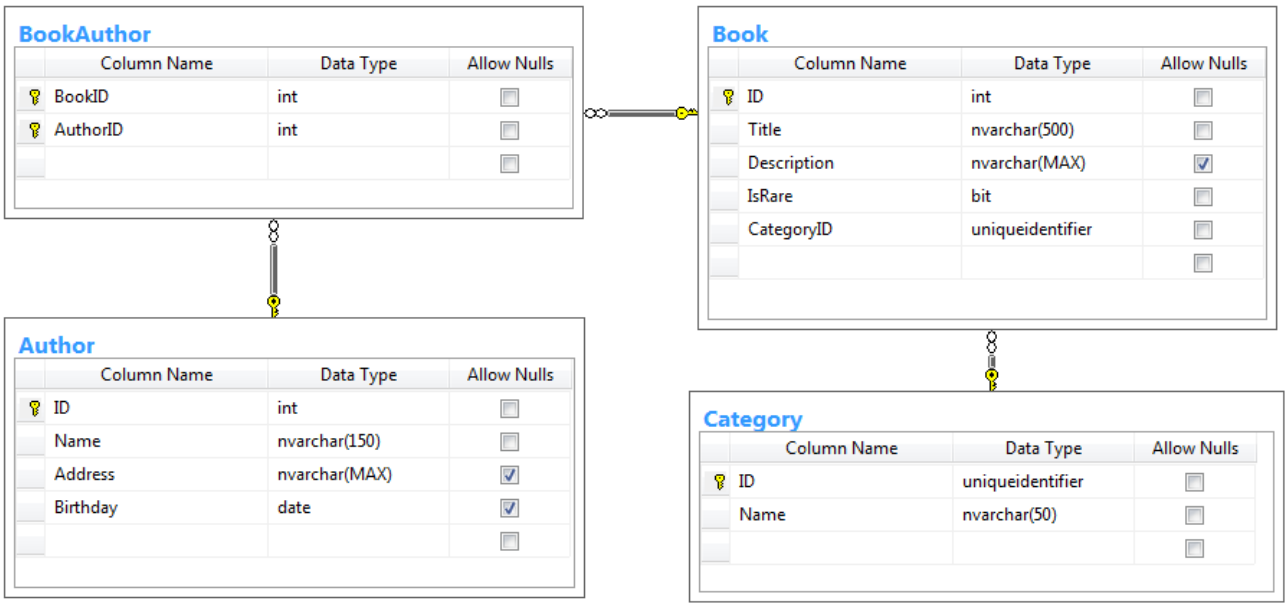

Рис. В.1. Схема базы данных, используемой в примерах

1. АРХИТЕКТУРА ADO.NET

Технология ADO.NET – это часть платформы Microsoft .NET Framework, обеспечивающая основные возможности по работе с реляционными базами данных и источниками данных. Рассмотрим общую архитектуру ADO.NET (рис. 1.1). Главными элементами ADO.NET являются: поставщик данных, абстрактная фабрика поставщиков и рассоединённый набор данных.

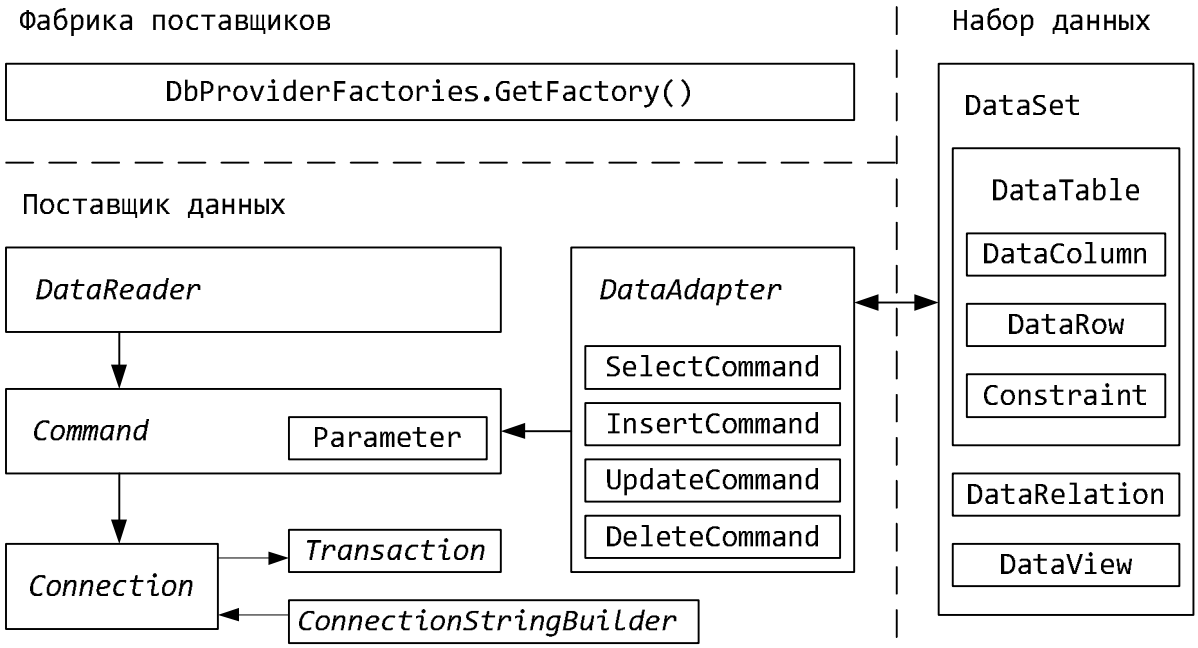

Рис. 1.1. Архитектура ADO.NET

Поставщик данных (*data provider*) – это совокупность типов для непосредственного взаимодействия с конкретным источником данных. Любой поставщик содержит четыре основных класса: `Connection`, `Command`, `DataReader` и `DataAdapter`[1]. Назначение `Connection` – установка и поддержка соединения с источником данных. `Command` служит для выполнения команд и запросов. Можно выполнить команды, не возвращающие данные (например создать в базе таблицу), и запросы для получения скалярного значения или набора данных. В последнем случае для чтения данных используется `DataReader` – однонаправленный курсор в режиме «только для чтения». `DataAdapter` служит своеобразным «мостом» между поставщиком данных и рассоединённым набором данных. Этот класс содержит четыре команды для выборки, обновления, вставки и удаления данных.

В состав поставщика данных входят некоторые вспомогательные элементы. Класс `ConnectionStringBuilder` используется для построения строк соединения. Класс `Transaction` служит для описания транзакции данных. Класс `Parameter` описывает параметры отдельной команды, а класс `CommandBuilder`

[1] Имена выделены курсивом, так как это условные названия, а не конкретные классы

облегчает построение самих команд. Также поставщик имеет классы *Exception* и *Error* для описания исключительных ситуаций и ошибок, возникающих при взаимодействии с источником данных.

Типы поставщика данных специфичны для каждого источника данных (в частности для конкретной СУБД) и находятся в изолированных пространствах имён и сборках. Имена типов поставщика обычно имеют общий префикс. Унификация поставщиков достигается благодаря тому, что их основные классы реализуют стандартные интерфейсы и наследуются от стандартных предков. Например, любой класс *Connection* реализует интерфейс `IDbConnection` и наследуется от `DbConnection`. Стандартные базовые элементы содержатся в пространстве имён `System.Data.Common`.

Вместе с .NET Framework инсталлируются поставщики для наиболее распространённых источников данных:

– *поставщик данных для SQL Server*. Предоставляет доступ к данным для Microsoft SQL Server 7.0 или более поздних версий. Использует пространство имён `System.Data.SqlClient`. Префикс имён типов – *Sql* (например `SqlConnection`);

– *поставщик данных для OLE DB*. Предназначен для источников данных OLE DB. Использует пространство имён `System.Data.OleDb`. Префикс имён типов – *OleDb* (например `OleDbCommand`);

– *поставщик данных для ODBC*. Предназначен для источников данных ODBC. Использует пространство имён `System.Data.Odbc`. Префикс имён типов – *Odbc* (например `OdbcDataReader`);

– *поставщик данных для Oracle*. Предоставляет доступ к СУБД Oracle версии 8.1.7 и старше и использует пространство имён `System.Data.OracleClient`. Префикс имён типов – *Oracle* (например `OracleException`);

– *поставщик EntityClient*. Предоставляет доступ к данным для приложений модели EDM (Entity Data Model). Использует пространство имён `System.Data.EntityClient`. Префикс имён типов – *Entity* (например `EntityParameter`).

В ADO.NET реализована модель получения экземпляров поставщиков, основанная на шаблоне проектирования *абстрактная фабрика*. Класс `DbProviderFactory` – предок для фабрики классов конкретного поставщика (например для фабрики `SqlClientFactory`). Класс `DbProviderFactories` содержит статический метод `GetFactory()` для получения фабрики поставщика по строковому имени (обычно используется название пространства имён поставщика):

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

Последним элементом архитектуры ADO.NET является *рассоединённый набор данных* (*data set*). Набор данных содержит информационный фрагмент источника данных. Для описания набора данных используются классы из пространства имён `System.Data`. Эти классы универсальны и не зависят от поставщика данных. Главным компонентом набора является класс `DataSet`, агрегирующий объекты остальных классов. Класс `DataTable` служит для описания таблиц. Элементами класса `DataTable` являются коллекции объектов `DataColumn` (колонки таблицы), `DataRow` (строки таблицы) и `Constraint` (ограничения на значения элементов таблицы). Класс `DataRelation` описывает связи между таблицами.

2. СОЕДИНЕНИЕ С ИСТОЧНИКОМ ДАННЫХ

В ADO.NET любое действие с источником данных можно выполнить при наличии соединения с этим источником. Каждый поставщик данных содержит класс, ответственный за описание соединений. Например, поставщик данных для SQL Server использует класс `SqlConnection`.

Класс соединения реализует интерфейс `System.Data.IDbConnection`. и наследуется от класса `System.Data.Common.DbConnection`. Наиболее важные элементы класса `DbConnection` перечислены в табл. 2.1.

Таблица 2.1

Некоторые элементы класса `DbConnection`

Имя элемента	Описание
`BeginTransaction()` и `EnlistTransaction()`	Методы позволяют начать транзакцию, связанную с соединением, или прикрепить соединение к транзакции
`ChangeDatabase()`	Устанавливает новую базу данных для использования
`Close()` и `Dispose()`	Методы выполняют закрытие активного соединения
`ConnectionString`	Строка с параметрами подключения к источнику данных
`ConnectionTimeout`	Время ожидания открытия соединения (в секундах), по истечении которого попытка соединения завершается и генерируется ошибка
`CreateCommand()`	Возвращает объект команды, специфичный для конкретного поставщика данных
`Database`	Имя текущей базы данных после открытия соединения или имя базы данных, указанное в строке подключения
`DataSource`	Имя сервера БД, к которому осуществляется подключение
`GetSchema()`	Возвращает схему источника данных, к которому выполнено подключение, в виде объекта `DataTable`
`Open()`	Метод выполняет попытку соединения с источником данных
`ServerVersion`	Строка с версией сервера базы данных
`State`	Состояние соединения (из перечисления `ConnectionState`)
`StateChange`	Событие, генерируемое при изменении состояния соединения

Конкретные поставщики данных могут добавлять или переопределять элементы `DbConnection`. Класс `SqlConnection` имеет свойство `PacketSize` (размер пакета обмена с сервером в байтах), свойство `StatisticsEnabled` (управление сбором статистики для текущего соединения), свойство `WorkstationId` (строковый идентификатор подключившегося клиента). Также `SqlConnection` поддерживает событие `InfoMessage`, генерируемое, если сервер БД послал клиенту строку с предупреждением или ошибкой.

Для задания параметров подключения объект соединения использует *строку подключения*, в которой через точку с запятой перечислены пары вида `имя параметра=значение`. Каждый поставщик данных может использовать специфичные имена параметров[1]. В табл. 2.2 приведены основные параметры, допустимые для поставщика SQL Server.

Таблица 2.2

Основные параметры строки подключения для SQL Server

Имя параметра, возможные синонимы	Значение по умолчанию	Описание параметра
`Asynchronous Processing`, `Async`	`false`	Включает или выключает поддержку асинхронных операций с базой данных
`AttachDBFilename`, `Initial File Name`		Имя базы данных, представленной отдельным файлом с расширением `.mdf`
`Connect Timeout`, `Connection Timeout`	`15`	Время ожидания открытия соединения (в секундах)
`Connection Lifetime`	`0`	Время жизни открытого соединения в пуле (в секундах). Значение `0` означает, что `Connection Lifetime = Connect Timeout`
`Current Language`		Естественный язык, используемый сервером в операциях сортировки, поиска и т. п.
`Data Source`, `Server`, `Address`, `Addr`, `Network Address`		Имя или сетевой адрес сервера баз данных
`Initial Catalog`, `Database`		Имя базы данных
`Integrated Security`, `Trusted_Connection`	`false`	Если установлено в `true` или `SSPI`, поставщик данных пытается подключиться к серверу, используя имя и пароль пользователя в Windows
`Max Pool Size`	`100`	Максимальное число соединений в пуле
`Min Pool Size`	`0`	Минимальное число соединений в пуле
`Network Library`, `Net`	`dbmssocn`	Имя сетевой библиотеки, которая используется для подключения к SQL Server
`Packet Size`	`8000`	Размер сетевого пакета (в байтах) для обмена с сервером
`Password`, `Pwd`		Пароль пользователя базы данных

[1] Сайт http://www.connectionstrings.com/ содержит множество примеров строк подключения

Окончание табл. 2.2

Имя параметра, возможные синонимы	Значение по умолчанию	Описание параметра
`Persist Security Info`	`false`	При установке в `false` критическая в плане безопасности информация (например пароль) удаляется из свойства `ConnectionString` сразу после осуществления подключения
`Pooling`	`true`	Булево выражение, определяет необходимость использования пула соединений
`User ID`		Идентификатор пользователя базы данных
`Workstation ID`		Имя компьютера, подключающегося к серверу

Для поставщика SQL Server обязательным является задание в строке подключения параметров `Data Source` и `Initial Catalog`, а также данных аутентификации. Можно применить два вида аутентификации. Если база использует встроенную аутентификацию, то передаётся имя пользователя и пароль. Сервер SQL Server может применять интегрированную Windows-аутентификацию. Тогда в строке подключения указывается `Integrated Security=SSPI`.

Задать строку подключения можно, либо указав её как параметр конструктора класса соединения, либо при помощи свойства `ConnectionString`. Естественно, строка подключения задаётся до вызова у соединения метода `Open()`.

Рассмотрим пример кода, в котором создаётся и открывается соединение для поставщика данных SQL Server:

```
string cs = @"Data Source=(local)\SQLExpress;" +
             "Initial Catalog=Library;" +
             "User ID=John Doe;Password=123";
var connection = new SqlConnection(cs);
connection.Open();
```

Пример демонстрирует очевидное неудобство, возникающее при работе со строками подключения. Параметры строки необходимо помнить и писать без ошибок, так как эти ошибки будут обнаружены только в момент подключения к источнику, но не при компиляции. В поставщиках данных имеется специальный класс, унаследованный от `DbConnectionStringBuilder`, который предназначен для построения правильных строк подключения. Типизированные свойства этого класса соответствуют отдельным параметрам строки подключения. Пример демонстрирует использование класса `SqlConnectionStringBuilder`:

```
var builder = new SqlConnectionStringBuilder();
builder.DataSource = @"(local)\SQLExpress";
builder.InitialCatalog = "Library";
builder.IntegratedSecurity = true;
var connection = new SqlConnection(builder.ConnectionString);
```

Серверы баз данных устанавливают лимит на количество одновременных подключений. Поэтому открытые соединения следует обязательно закрывать после использования. Для этого применяется метод `Close()` или метод `Dispose()`, являющийся реализацией интерфейса `IDisposable`:

```
// использование using гарантирует вызов connection.Dispose()
using(var connection = new SqlConnection(builder.ConnectionString))
{
    connection.Open();
    // работа с соединением
}
```

Для увеличения производительности поставщики данных могут поддерживать *пул соединений* (*connection pool*). Сущность пула заключается в следующем. При вызове методов `Close()` или `Dispose()` соединение с базой не разрывается, а помещается в буфер. Если приложение захочет открыть соединение, аналогичное существующему в буфере, то система возвращает открытое подключение из пула[1]. Какие подключения считаются «аналогичными», зависит от поставщика данных. Например, поставщик для SQL Server требует буквального совпадения строк подключения с точностью до символа. Настройка пула соединений выполняется при помощи параметров строки подключения (`Pooling`, `Max\Min Pool Size`, `Connection Lifetime`).

В заключение рассмотрим вопросы, связанные с обработкой ошибок при работе с источниками данных. Обычно при возникновении ошибки источника на клиенте генерируется исключительная ситуация особого типа, специфичного для каждого поставщика данных. Например, для поставщика SQL Server это исключение типа `SqlException`. В объекте `SqlException` доступно свойство `Errors` – набор объектов типа `SqlError`. В этих объектах содержится дополнительная информация об ошибке:

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

[1] Пул существует в рамках одного домена (разные домены приложения имеют разные пулы)

3. ВЫПОЛНЕНИЕ КОМАНД И ЗАПРОСОВ К ИСТОЧНИКУ ДАННЫХ

Работа с источником данных основана на выполнении запросов и команд. Для поддержки команд каждый поставщик данных предоставляет класс, который реализует интерфейс `IDbCommand` и наследуется от класса `DbCommand`. Основные элементы класса `DbCommand` приведены в табл. 3.1.

Таблица 3.1

Основные элементы класса `DbCommand`

Имя элемента	Описание
`Cancel()`	Метод пытается отменить выполнение команды
`CommandText`	Текст команды (строка)
`CommandTimeout`	Время ожидания (в секундах) перед завершением попытки выполнить команду и созданием ошибки
`CommandType`	Указывает способ интерпретации свойства `CommandText`
`Connection`	Соединение, используемое командой
`CreateParameter()`	Метод создаёт новый объект, представляющий параметр команды
`ExecuteNonQuery()`	Выполняет команду, которая не возвращает данные из источника
`ExecuteReader()`	Метод выполняет команду и возвращает объект `DbDataReader`
`ExecuteScalar()`	Выполняет запрос и возвращает первый столбец первой строки результирующего набора (другие столбцы и строки игнорируются)
`Parameters`	Коллекция параметров команды
`Prepare()`	Создаёт подготовленную (скомпилированную) версию команды
`Transaction`	Транзакция `DbTransaction`, внутри которой выполняется команда
`UpdatedRowSource`	Способ применения результатов команды к объекту `DataRow`, если команда используется методом `Update()` объекта `DbDataAdapter`

Рассмотрим использование класса команды `SqlCommand`, описанного в поставщике данных для SQL Server. Класс `SqlCommand` содержит несколько перегруженных конструкторов. В частности, можно создать команду, указав текст команды и соединение, с которым связана команда:

```
// конструктор без параметров
var cmd1 = new SqlCommand();

// задаём текст команды
var cmd2 = new SqlCommand("SELECT * FROM Book");

// указываем текст команды и соединение
var cmd3 = new SqlCommand("SELECT * FROM Book", connection);
```

Объект команды также может быть создан при помощи фабрики поставщика или на основе объекта-соединения (в последнем случае команда автоматически ассоциируется с соединением):

```
DbCommand cmd4 = SqlClientFactory.Instance.CreateCommand();

SqlConnection connection = new SqlConnection();
SqlCommand cmd5 = connection.CreateCommand();
```

Свойство команды `CommandText` содержит текст команды, а свойство `CommandType` определяет, как следует понимать этот текст. `CommandType` может принимать следующие значения:

– `CommandType.Text`. Текст команды – это SQL-инструкции. Этот тип команды устанавливается по умолчанию;

– `CommandType.StoredProcedure`. Текст команды – имя хранимой процедуры, которая находится в базе данных;

– `CommandType.TableDirect`. Команда предназначена для извлечения из БД полной таблицы. Имя таблицы указывается в `CommandText`. Данный тип команд поддерживает только поставщик данных для OLE DB.

Свойство команды `CommandTimeout` задаёт время, в течение которого ожидается начало выполнения команды (при использовании `SqlCommand` значение этого свойства по умолчанию – 30 с). Следует учитывать, что после начала выполнения команды данное свойство никакой роли не играет. Выполнение команды не прервётся, даже если она будет получать данные из базы на протяжении, например, одной минуты.

Перед выполнением любая команда должна быть связана с соединением (свойство `Connection`), и это соединение должно быть открыто. В ADO.NET существует несколько способов выполнения команд, которые отличаются лишь информацией, возвращаемой из БД. Ниже перечислены методы выполнения команд, поддерживаемые всеми поставщиками:

– `ExecuteNonQuery()`. Метод применяется для запросов, не возвращающих данные. Метод возвращает суммарное число строк, добавленных, изменённых или удалённых в результате выполнения команды;

– `ExecuteScalar()`. Метод выполняет команду и возвращает первый столбец первой строки первого результирующего набора данных. Метод может быть полезен при выполнении запросов или хранимых процедур, возвращающих единственный результат;

– `ExecuteReader()`. Метод выполняет команду и возвращает *`DataReader`* – однонаправленный курсор данных в режиме «только для чтения». Тип возвращаемого объекта соответствует поставщику. Метод используется, когда требуется получить набор данных.

Приведём пример использования метода `ExecuteScalar()`:

```
using (var connect = new SqlConnection(builder.ConnectionString))
{
    var command = connect.CreateCommand();
    command.CommandText = "SELECT COUNT(*) FROM Book";
    connect.Open();

    // приведение типов, так как ExecuteScalar() возвращает object
    var count = (int)command.ExecuteScalar();
}
```

В дополнение к основным методам выполнения команд класс SqlCommand поддерживает метод ExecuteXmlReader(), возвращающий объект XmlReader. Метод работает с SQL Server 2000 или более поздней версией этой СУБД и требует, чтобы получаемые из базы данные имели формат XML. Кроме этого, класс SqlCommand реализует возможность асинхронных операций с базой. У методов ExecuteNonQuery(), ExecuteReader() и ExecuteXmlReader() имеются парные асинхронные аналоги (см. пример кода ниже). Для включения поддержки асинхронных операций необходимо задать в строке соединения параметр Asynchronous Processing=true.

```
IAsyncResult pending = command.BeginExecuteNonQuery();
while (pending.IsCompleted == false)
{
    // имитируем выполнение полезной работы
    System.Threading.Thread.Sleep(100);
}
command.EndExecuteNonQuery(pending);
```

4. ПАРАМЕТРИЗИРОВАННЫЕ КОМАНДЫ

Команда, выполняемая над источником данных, может содержать заданные пользователем параметры. Например, в случае использования поставщика данных для SQL Server и команды с типом CommandType.Text следующая строка определяет запрос для получения из таблицы Book всех значений колонки Name таких записей, у которых колонка ID равна параметру @id:

```
SELECT Name FROM Book WHERE ID = @id
```

Для работы с параметрами поставщики данных определяют особые классы, унаследованные от DbParameter. Некоторые свойства класса DbParameter перечислены ниже (не каждый параметр требует задания всех этих свойств):

– ParameterName. Имя параметра. У поставщика для SQL Server имя любого параметра предваряется символом @. Другие поставщики могут не использовать имена, а определять параметры по позиции в тексте команды;

– DbType. Тип данных параметра. Перечисление System.Data.DbType содержит элементы, которые могут использоваться как значения этого свойства. Кроме этого, каждый поставщик имеет перечисления либо классы, более точно отражающие реальные типы источника данных;

– Size. Свойство зависит от типа данных параметра и обычно используется для указания максимальной длины данных. Например, для строковых типов свойство задаёт максимальный размер строки. Значение по умолчанию определяется по свойству DbType. В случае числовых типов изменять это значение не требуется;

– Direction. Данное свойство определяет способ передачи параметра хранимой процедуры. Его возможные значения – Input, Output, InputOutput

и ReturnValue – представлены перечислением ParameterDirection. По умолчанию используется значение Input;

– IsNullable. Это свойство определяет, может ли параметр принимать значения NULL. По умолчанию свойство установлено в false;

– Value. Значение параметра. Для параметров Input и InputOutput это свойство должно быть установлено до выполнения команды, для параметров Output, InputOutput и ReturnValue значение свойства устанавливается в результате выполнения команды. Чтобы передать пустой входной параметр, нужно либо не устанавливать значение свойства Value, либо установить его равным DBNull.Value;

– SourceColumn и SourceVersion. Эти свойства определяют способ использования параметра с адаптером данных и рассматриваются ниже.

В дополнение к приведённому списку класс SqlParameter имеет свойства Precision, Scale, LocaleID и некоторые другие. Кроме этого, SqlParameter использует для типов данных параметра перечисление System.Data.SqlDbType (определяя свойства SqlDbType и SqlValue). В табл. 4.1 приведено соответствие типов SQL Server и элементов DbType и SqlDbType.

Таблица 4.1

Соответствие типов при работе с SQL Server

Тип SQL Server	Тип .NET Framework	Элемент SqlDbType	Элемент DbType
bigint	Int64	BigInt	Int64
binary	Byte[]	VarBinary	Binary
bit	Boolean	Bit	Boolean
char	String, Char[]	Char	AnsiStringFixedLength, String
date	DateTime	Date	Date
datetime	DateTime	DateTime	DateTime
datetime2	DateTime2	DateTime2	DateTime2
datetimeoffset	DateTimeOffset	DateTimeOffset	DateTimeOffset
decimal	Decimal	Decimal	Decimal
varbinary(max)	Byte[]	VarBinary	Binary
float	Double	Float	Double
image	Byte[]	Binary	Binary
int	Int32	Int	Int32
money	Decimal	Money	Decimal
nchar	String, Char[]	NChar	StringFixedLength
ntext	String, Char[]	NText	String
numeric	Decimal	Decimal	Decimal
nvarchar	String, Char[]	NVarChar	String
real	Single	Real	Single
rowversion	Byte[]	Timestamp	Binary
smalldatetime	DateTime	DateTime	DateTime

Окончание табл. 4.1

Тип SQL Server	Тип .NET Framework	Элемент SqlDbType	Элемент DbType
smallint	Int16	SmallInt	Int16
smallmoney	Decimal	SmallMoney	Decimal
sql_variant	Object	Variant	Object
text	String, Char[]	Text	String
time	TimeSpan	Time	Time
timestamp	Byte[]	Timestamp	Binary
tinyint	Byte	TinyInt	Byte
uniqueidentifier	Guid	UniqueIdentifier	Guid
varbinary	Byte[]	VarBinary	Binary
varchar	String, Char[]	VarChar	AnsiString, String
xml	Xml	Xml	Xml

Для создания и настройки параметра можно использовать один из конструкторов класса, описывающего параметр, или применить синтаксис упрощённой инициализации объектов:

```
var cmd = new SqlCommand("SELECT Name FROM Book WHERE ID = @id");
var p = new SqlParameter();
p.ParameterName = "@id";
p.Direction = ParameterDirection.Input;
p.SqlDbType = SqlDbType.Int;
```

Любой класс команды содержит свойство `Parameters`, представляющее коллекцию параметров. Параметр помещается в эту коллекцию при помощи метода `Add()`, имеющего несколько перегруженных версий. Для доступа к параметрам в коллекции используется строковый индекс (имя параметра) или целочисленный индекс (позиция параметра).

Следующий пример показывает выборку данных из таблицы с использованием параметризированного запроса:

```
var cmd = new SqlCommand("SELECT Name FROM Book WHERE ID = @id");
cmd.Connection = new SqlConnection(builder.ConnectionString);
cmd.Parameters.Add("@id", SqlDbType.Int);

Console.Write("Input ID: ");
cmd.Parameters["@id"].Value = int.Parse(Console.ReadLine());

cmd.Connection.Open();
Console.WriteLine((string)cmd.ExecuteScalar());
cmd.Connection.Close();
```

Поставщик для SQL Server позволяет вывести информацию о параметрах команды из хранимой процедуры, для вызова которой используется команда.

Статический метод DeriveParameters() класса SqlCommandBuilder автоматически заполняет коллекцию параметров команды:

```
var cmd = new SqlCommand("Select_Procedure");
cmd.Connection = new SqlConnection(builder.ConnectionString);
cmd.CommandType = CommandType.StoredProcedure;

SqlCommandBuilder.DeriveParameters(cmd);
```

Заметим, что параметризированные запросы зачастую удобнее сформировать, воспользовавшись функциями для работы со строками (и такой метод более быстрый, чем работа с объектами-параметрами). Однако использование параметров вместо простой конкатенации строк позволяет писать более безопасный код, в меньшей степени подверженный атакам типа *Sql Injection*. Кроме этого, без параметров не обойтись, если речь заходит о вызове параметризированной хранимой процедуры.

5. ЧТЕНИЕ НАБОРА ДАННЫХ И ОБЪЕКТ DATAREADER

Получение набора данных из источника данных – одна из наиболее часто выполняемых операций. Для получения набора данных классы команд содержат метод ExecuteReader(), возвращающий объект *DataReader*. Объект *DataReader* имеет следующие особенности. Он позволяет перемещаться по данным набора строго последовательно и в одном направлении – от начала к концу. Данные, полученные при помощи *DataReader*, доступны только для чтения. И, наконец, на время чтения данных соответствующее соединение с базой блокируется, то есть соединение не может быть использовано другими командами, пока чтение данных не завершено.

Каждый поставщик имеет собственный класс для *DataReader*, но любой такой класс реализует интерфейсы IDataRecord, IDataReader, IDisposable, IEnumerable и наследуется от DbDataReader. Элементы интерфейсов IDataRecord и IDataReader перечислены в табл. 5.1, 5.2 соответственно (IDataReader наследуется от IDataRecord).

Таблица 5.1

Элементы интерфейса IDataRecord

Имя элемента	Описание
GetBoolean(), GetByte(), GetBytes(), GetChar(), GetChars(), GetDateTime(), GetDecimal(), GetDouble(), GetFloat(), GetGuid(), GetInt16(), GetInt32(), GetInt64(), GetString()	Методы получают значение поля по указанному индексу в виде данных соответствующего типа
GetData()	Возвращает IDataReader для заданного порядкового номера поля

Окончание табл. 5.1

Имя элемента	Описание
`GetDataTypeName()`	Получает имя типа для поля, указанного по индексу
`GetFieldType()`	Получает объект `Type` для поля, указанного по индексу
`GetName()`	Получает имя для поля, указанного по индексу
`GetOrdinal()`	Возвращает индекс поля по имени поля
`GetValue()`	Возвращает значение поля, указанного по индексу, как `object`
`GetValues()`	Заполняет массив объектов значениями полей текущей записи
`FieldCount`	Количество полей в текущей строке
`Item[int]`	Получает поле, расположенное по указанному индексу
`Item[string]`	Получает поле с указанным именем
`IsDBNull()`	Возвращает сообщение о том, имеет ли указанное поле значение `NULL`

Таблица 5.2

Элементы интерфейса `IDataReader`

Имя элемента	Описание
`Close()`	Метод закрывает объект `IDataReader`
`Depth`	Глубина вложенности для текущей строки
`GetSchemaTable()`	Возвращает объект `DataTable`, описывающий метаданные столбца объекта `IDataReader`
`IsClosed`	Булевское значение – закрыт ли объект чтения данных
`NextResult()`	Вызывает переход считывателя данных к следующему результату при выполнении пакетных операторов SQL
`Read()`	Перемещает `IDataReader` к следующей записи
`RecordsAffected`	Количество строк, которые были изменены, вставлены или удалены при выполнении инструкции SQL

Основным методом объекта *`DataReader`* является метод `Read()`, который перемещает указатель на следующую запись в наборе данных и возвращает значение `false`, если записей в наборе больше нет. После прочтения всех записей у объекта *`DataReader`* необходимо вызвать метод `Close()` для освобождения соединения, занятого *`DataReader`*. Рассмотрим шаблон использования объекта *`DataReader`* на примере `SqlDataReader`:

```
// стандартные подготовительные действия
var connection = new SqlConnection(builder.ConnectionString);
var command = new SqlCommand("SELECT * FROM Book", connection);

// выполняем команду и получаем объект DataReader
connection.Open();
var reader = command.ExecuteReader();
```

```
// читаем данные в цикле
while (reader.Read())
{
    // здесь размещается код чтения полей записи
}

// закрываем объект DataReader и соединение
reader.Close();
connection.Close();
```

Чтобы прочитать в цикле поля отдельной записи, можно воспользоваться индексатором *DataReader*, где в качестве аргумента выступает строка с именем поля. Индексатор имеет тип `object`, поэтому необходимо выполнять приведение типов. Поиск поля по имени не зависит от регистра[1]. Если нужного поля в наборе данных нет, генерируется исключение `IndexOutOfRangeException`.

```
while (reader.Read())
{
    int id = (int) reader["ID"];
    string title = (string)reader["Title"];
}
```

Поиск поля по имени требует сравнения строк и происходит медленно. Альтернативой является использование индексатора, у которого в качестве аргумента применяется номер поля. Теоретически это уменьшает гибкость приложения. Однако порядок столбцов в наборе данных меняется редко (если меняется команда запроса или структура объекта базы). В большинстве приложений можно без каких-либо проблем жёстко задать индексы всех полей. Если известно только имя поля, метод объекта *DataReader* `GetOrdinal()` может вернуть индекс поля по имени. Это позволяет достичь компромисса между гибкостью и производительностью:

```
// выполняем команду и получаем объект DataReader
var reader = command.ExecuteReader();

// один раз находим номер столбца по имени
int idIndex = reader.GetOrdinal("ID");
int titleIndex = reader.GetOrdinal("Title");

// в цикле доступ будет быстрее
while (reader.Read())
{
    int id = (int)reader[idIndex];
    string title = (string)reader[titleIndex];
}
```

[1] Вначале производится поиск соответствия с учетом регистра, если соответствие не найдено – повторный поиск без учета регистра

Объект *DataReader* имеет ряд методов вида Get*ТипДанных*() (например GetInt32()). Эти методы получают в качестве параметра индекс поля, а возвращают значение поля, приведенное к соответствующему типу .NET. Класс SqlDataReader дополнительно определяет методы вида Get*SqlТипДанных*() (например GetSqlInt32()). Такие методы возвращают значение поля в виде одной из структур, описанных в пространстве имён System.Data.SqlTypes.

```
while (reader.Read())
{
    int id = reader.GetInt32(idIndex);

    // SqlString содержит набор полезных элементов
    SqlString titleStruct = reader.GetSqlString(titleIndex);
    string title = titleStruct.Value;
}
```

Отдельные поля записи могут иметь значения NULL, то есть быть незаполненными. При попытке извлечь значения из NULL-поля (точнее при попытке преобразования значения в требуемый тип) будет сгенерировано исключение. Объект *DataReader* содержит метод IsDBNull(), предназначенный для индикации пустых полей:

```
if (!reader.IsDBNull(idIndex))
{
    int id = reader.GetInt32(idIndex);
}
```

Отдельные поставщики позволяют выполнить запрос к данным, возвращающий несколько результирующих наборов. При наличии такой возможности объект *DataReader* реализует метод NextResult(), который выполняет переход к следующему набору или возвращает значение false, если такого набора нет. Рассмотрим пример кода для SqlDataReader:

```
// отсутствует код создания соединения и команды
// запрос возвращает два набора данных
command.CommandText = "SELECT * FROM Book;  SELECT * FROM Author";
connection.Open();
var reader = command.ExecuteReader();

// вложенные циклы, внешний цикл – по наборам данных
do
{
    while (reader.Read())
    {
        int id = reader.GetInt32(0);
    }
}
while (reader.NextResult());
```

Разберём оставшиеся свойства и методы объекта *DataReader*. Свойство FieldCount возвращает количество полей отдельной записи в наборе данных. Булевское свойство HasRows помогает выяснить, имеет ли набор данных хотя бы одну запись. Булевское свойство IsClosed показывает, закрыт ли объект *DataReader*. Свойство RecordsAffected позволяет определить число записей, измененных запросом[1]. Метод GetValues() позволяет поместить содержимое записи в массив. Если нужно максимально быстро получить содержимое каждого поля, использование метода GetValues() обеспечит более высокую производительность, чем чтение значений отдельных полей.

```
var reader = command.ExecuteReader();
var data = new object[reader.FieldCount];
while (reader.Read())
{
    // на самом деле GetValues() – функция,
    // которая возвращает количество полей прочитанной записи
    reader.GetValues(data);
    int id = (int)data[0];
    string title = (string)data[1];
}
```

Для исследования структуры возвращаемого набора данных можно применить метод объекта *DataReader* GetSchemaTable(). Этот метод создаёт объект DataTable, строки которого описывают столбцы полученного набора данных. Колонки таблицы соответствуют атрибутам этих столбцов[2]. Следующий пример кода выводит для каждого столбца его имя и тип:

```
var reader = command.ExecuteReader();
DataTable table = reader.GetSchemaTable();
foreach (DataRow row in table.Rows)
{
    Console.WriteLine(row["ColumnName"] + " - " + row["DataType"]);
}
```

Вернёмся к методу команды ExecuteReader(). Этот метод перегружен и может принимать значения из перечисления флагов CommandBehavior, которые указаны в табл. 5.3 (допустимо использовать комбинацию значений).

---

[1] Это имеет смысл в том случае, если выполняется команда, которая комбинирует несколько SQL-операторов. Например, команда с таким текстом:

```
SELECT * FROM Book; DELETE FROM Book WHERE ID = 1
```

[2] Для поставщика SQL Server таблица с описанием набора содержит столбцы: ColumnName, ColumnOrdinal, ColumnSize, NumericPrecision, NumericScale, IsUnique, IsKey, BaseServerName, BaseCatalogName, BaseColumnName, BaseSchemaName, BaseTableName, DataType, AllowDBNull, ProviderType, IsAliased, IsExpression, IsIdentity, IsAutoIncrement, IsRowVersion, IsHidden, IsLong, IsReadOnly

Таблица 5.3

Элементы перечисления `CommandBehavior`

Имя элемента	Описание
`Default`	Запрос может вернуть несколько наборов, выполнение запроса может повлиять на состояние базы данных (вызов `ExecuteReader(CommandBehavior.Default)` эквивалентен вызову `ExecuteReader()`)
`SingleResult`	Запрос возвращает один набор результатов
`SchemaOnly`	Запрос возвращает только сведения о столбцах набора результатов
`KeyInfo`	Запрос возвращает дополнительную информацию для схемы, чтобы показать, являются ли столбцы набора ключевыми столбцами таблиц
`SingleRow`	Ожидается, что запрос вернет одну строку из первого набора результатов. Некоторые поставщики данных могут использовать эту информацию для оптимизации производительности команды
`SequentialAccess`	Указывает для объекта *`DataReader`* способ обработки строк, содержащих поля с большими двоичными значениями. Вместо загрузки всего поля позволяет *`DataReader`* загрузить данные поля как поток. Затем можно использовать методы `GetBytes()` или `GetChars()`, чтобы указать положение указателя в потоке для начала операции чтения и размер буфера для возврата данных
`CloseConnection`	При выполнении команды связанный объект *`Connection`* закрывается, когда закрывается соответствующий объект *`DataReader`*

6. РАБОТА С ТРАНЗАКЦИЯМИ В ADO.NET

Транзакция – это набор операций, которые для обеспечения целостности и корректного поведения системы должны быть выполнены успешно или неудачно только все вместе. Транзакции характеризуются четырьмя свойствами, часто называемыми *свойства ACID*:

– *Неделимость* (*Atomic*). Все шаги транзакции должны быть выполнены или не выполнены вместе. До тех пор пока все шаги транзакции не будут завершены, транзакция не считается оконченной;

– *Согласованность* (*Consistent*). Транзакция переводит набор данных из одного стабильного состоянии в другое;

– *Изолированность* (*Isolated*). Транзакция не должна влиять на другую транзакцию, выполняющуюся в то же время;

– *Долговечность* (*Durable*). Транзакция считается успешной только тогда, когда изменения, которые произошли во время транзакции, сохраняются на каком-то долговременном носителе, обычно на жестком диске.

Транзакции часто используются во многих бизнес-приложениях, потому что обеспечивают устойчивость и предсказуемость системы. Чтобы в таких программных системах можно было применить концепцию транзакций, транзакции должен поддерживать источник данных. Это делают практически все современные СУБД. Например, SQL Server 2008 обеспечивает реализацию таких операторов T-SQL, как `BEGIN TRANSACTION` (начать транзакцию), `SAVE`

TRANSACTION (сохранить транзакцию), COMMIT TRANSACTION (зафиксировать транзакцию) и ROLLBACK TRANSACTION (откатить транзакцию).

Транзакции можно разделить на две категории – локальные и распределённые. *Локальная транзакция* использует поддерживающий транзакции источник данных (например SQL Sever) и не выходит за рамки одного соединения. Когда все участвующие в транзакции данные содержатся в одной базе, сама база обеспечивает выполнение правил ACID. *Распределённая транзакция* охватывает несколько источников данных, и ей может потребоваться, например, читать сообщения с сервера очереди сообщений (Message Queue Server), выбирать данные из базы SQL Server и записывать их в базы других СУБД.

Технология ADO.NET поддерживает транзакции одиночной базы, которые отслеживаются на основе соединений. Поставщики данных имеют собственные реализации класса транзакций (например класс SqlTransaction). Все такие классы реализуют интерфейс IDbTransaction. Метод Commit() идентифицирует транзакцию как успешную. Если этот метод выполнился без ошибки, то все ожидающие изменения записываются в базу данных. Метод Rollback() помечает транзакцию как неудачную – все ожидающие изменения аннулируются, а база данных остаётся в прежнем состоянии. Получить транзакцию можно у открытого объекта-соединения, вызвав метод BeginTransaction(). Для связи объекта команды и транзакции используется свойство команды Transaction.

```
// connection - соединение, command1 и command2 - объекты-команды
connection.Open();
SqlTransaction transaction = connection.BeginTransaction();
command1.Transaction = transaction;
command2.Transaction = transaction;
try
{
    command1.ExecuteNonQuery();
    command2.ExecuteNonQuery();
    transaction.Commit();
}
catch (Exception)
{
    transaction.Rollback();
}
finally
{
    connection.Close();
}
```

Важной характеристикой транзакции является её *уровень изоляции*. Это степень видимости внутри транзакции изменений, выполненных за пределами этой транзакции. Иначе говоря, уровень изоляции определяет, насколько чувствительна транзакция к изменениям, выполненным другими транзакциями.

При выполнении транзакций несколькими пользователями одной базы могут возникнуть следующие проблемы:

– *«Грязное» чтение* (*Dirty reads*). Первый пользователь начинает транзакцию, изменяющую данные. В это время другой пользователь (или создаваемая им транзакция) извлекает частично изменённые данные, которые не являются верными;

– *Неповторяемое чтение* (*Non-repeatable reads*). Первый пользователь начинает транзакцию. В это время другой пользователь начинает и завершает другую транзакцию. Первый пользователь при повторном чтении данных (например, если в его транзакцию входит несколько одинаковых инструкций `SELECT`) получает другой набор данных;

– *Чтение фантомов* (*Phantom reads*). Первый пользователь начинает транзакцию, извлекающую данные из таблицы. В это время другой пользователь начинает и завершает транзакцию, вставляющую или удаляющую данные. Первый пользователь получает набор данных, содержащий удалённые или изменённые строки (фантомы).

Для решения описанных проблем вводятся различные уровни изоляции транзакций. Уровни, доступные в ADO.NET, описаны элементами перечисления `IsolationLevel`:

– `ReadUncommitted`. Транзакция может считывать данные, с которыми работают другие транзакции. Применение этого уровня изоляции может привести ко всем перечисленным выше проблемам;

– `ReadCommited`. Это стандартный уровень изоляции как для SQL Server, так и для Oracle. Транзакция не может считывать данные, с которыми работают другие транзакции. Применение этого уровня изоляции исключает проблему «грязного» чтения. Однако данные могут быть изменены до завершения транзакции;

– `RepeatableRead`. Транзакция не может считывать данные, с которыми работают другие транзакции. Другие транзакции также не могут считывать данные, с которыми работает эта транзакция. Применение этого уровня изоляции исключает все проблемы, кроме чтения фантомов;

– `Snapshot`. Этот уровень изоляции уменьшает вероятность установки блокировки строк, сохраняя копию данных, которые одно приложение может читать, в то время как другое модифицирует эти же данные. Другими словами, если транзакция `A` модифицирует данные, то транзакция `B` не увидит выполненные изменения. Но важно то, что транзакция `B` не будет заблокирована и будет читать снимок данных, сделанный перед началом транзакции `A`. В SQL Server изоляция `Snapshot` перед использованием должна быть разрешена на уровне базы данных. Это можно сделать с помощью следующей SQL-команды: `ALTER DATABASE Имя_базы SET ALLOW SNAPSHOT ISOLATION ON;`

– `Serializable`. Транзакция полностью изолирована от других транзакций. Применение этого уровня изоляции полностью исключает все проблемы.

Значения уровня изоляции могут быть заданы как параметр метода `BeginTransaction()` класса `DbConnection`. Узнать уровень изоляции можно с помощью свойства `IsolationLevel` объекта транзакции.

Коротко остановимся на возможностях работы с распределёнными транзакциями. В платформе .NET распределённые транзакции реализуются при помощи типов из пространства имён `System.Transactions`. Рассмотрим следующий простой пример. Пусть приложение работает с двумя базами данных. Первая база `Credits` содержит одну таблицу `Credit` с двумя столбцами `CreditID` и `CreditAmount`. Вторая база `Debits` содержит таблицу `Debit` из двух столбцов `DebitID` и `DebitAmount`. Код описывает две соответствующие друг другу команды вставки данных, объединённые в одну распределённую транзакцию.

```
using (var scope = new TransactionScope())
{
    using (var connection = new SqlConnection(conn_string_1))
    {
        SqlCommand command = connection.CreateCommand();
        command.CommandText =
                "INSERT INTO Credit (CreditAmount) VALUES (100)";
        connection.Open();
        command.ExecuteNonQuery();
    }
    using (var connection = new SqlConnection(conn_string_2))
    {
        SqlCommand command = connection.CreateCommand();
        command.CommandText =
                "INSERT INTO Debit (DebitAmount) VALUES (100)";
        connection.Open();
        command.ExecuteNonQuery();
    }
    scope.Complete();
}
```

Как показывает пример, транзакция оформляется с помощью блока `using`. Приложение создаёт новый экземпляр `TransactionScope`, который определяет включаемую в транзакцию часть кода. Весь код между конструктором `TransactionScope` и вызовом `Dispose()` для экземпляра `TransactionScope` заносится в распределённую транзакцию.

7. ЭЛЕМЕНТЫ РАССОЕДИНЁННОГО НАБОРА ДАННЫХ

Технология ADO.NET предоставляет возможность работы с рассоединённым набором данных. Такой набор реализуется объектом класса `DataSet`. Это реляционная структура, которая хранится в памяти. `DataSet` содержит коллекцию таблиц (объекты класса `DataTable`) и связей между таблицами (объекты класса `DataRelation`). Отдельная таблица хранит набор столбцов (объекты класса `DataColumn`), строк (объекты класса `DataRow`) и ограничений (объекты

классов, унаследованных от Constraint). Столбцы и ограничения описывают структуру таблицы, а строки содержат данные таблицы.

Технически отдельные компоненты DataSet хранятся в специализированных коллекциях. Например, DataSet содержит коллекции Tables и Relations. Таблица имеет коллекции Columns (для колонок), Rows (для строк), Constraints (для ограничений), ParentRelations и ChildRelations (для связей таблицы). Любая подобная коллекция обладает сходным набором свойств и методов. Коллекции имеют перегруженные индексаторы для обращения к элементу по номеру или по имени, методы добавления, поиска и удаления элементов. Методы добавления перегружены и обеспечивают как добавление существующего объекта, так и автоматическое создание соответствующего объекта перед помещением в коллекцию.

Опишем состав класса DataTable. Этот класс является сериализуемым и поддерживает сохранение данных в формате XML. Класс DataTable реализует интерфейс IListSource, который позволяет объекту возвратить список своих данных (это необходимо для связывания с визуальными элементами управления). DataTable имеет конструктор без параметров и конструктор, принимающий в качестве аргумента строковое имя таблицы.

Наиболее важные свойства класса DataTable приведены в табл. 7.1.

Таблица 7.1

Свойства класса DataTable

Имя свойства	Описание
CaseSensitive	Показывает, учитывается ли регистр при сравнении строк в таблице
ChildRelations	Возвращает коллекцию дочерних отношений для таблицы
Columns	Набор столбцов таблицы
Constraints	Набор ограничений, заданных для таблицы
DataSet	Рассоединённый набор данных, к которому принадлежит таблица
DefaultView	Указывает на представление по умолчанию (DataView) для таблицы
HasErrors	Указывает, содержит ли таблица ошибки
Locale	Свойство имеет тип CultureInfo и определяет региональные параметры, используемые таблицей при сравнении строк
MinimumCapacity	Служит для получения или установки исходного количества строк таблицы (по умолчанию – 50 строк)
Namespace	Пространство имён XML, используемое при экспорте и импорте данных таблицы в XML-формате
ParentRelations	Коллекция родительских отношений для таблицы
Prefix	Префикс пространства имён XML
PrimaryKey	Массив столбцов, формирующих первичный ключ таблицы
RemotingFormat	Позволяет указать формат данных при сериализации объекта DataTable – бинарный формат или XML
Rows	Набор строк таблицы
TableName	Строка с именем таблицы

Методы класса DataTable перечислены в табл. 7.2.

Таблица 7.2

Методы класса DataTable

Имя метода	Описание
AcceptChanges()	Фиксирует все изменения данных в строках таблицы, которые были проделаны с момента предыдущего вызова AcceptChanges()
BeginLoadData()	Отключает все ограничения при загрузке данных
Clear()	Уничтожаются все строки таблицы
Clone()	Клонирует структуру таблицы и возвращает таблицу без строк
Compute()	Метод применяет строку-выражение, заданную в качестве параметра, к диапазону строк таблицы
Copy()	Метод клонирует и структуру, и данные таблицы
CreateDataReader()	Возвращает объект DataTableReader, обеспечивающий итерацию строк в текущей таблице
EndLoadData()	Активирует ограничения после загрузки данных
GetChanges()	Метод возвращает таблицу с идентичной схемой, содержащую изменения, которые еще не зафиксированы методом AcceptChanges()
GetErrors()	Возвращает массив объектов DataRow, которые нарушают ограничения таблицы
ImportRow()	В таблицу вставляется строка DataRow, передаваемая как параметр
Load()	Заполняет таблицу значениями из источника данных с помощью объекта IDataReader, передаваемого как параметр
LoadDataRow()	Добавляет или обновляет строку таблицы, основываясь на содержимом массива-параметра
Merge()	Объединяет заданный объект DataTable с текущей таблицей
NewRow()	Создается пустая строка DataRow по схеме столбцов таблицы
ReadXml()	Читает содержимое DataTable в XML-формате из файла, Stream, TextReader или XmlReader
ReadXmlSchema()	Работает как ReadXml(), но читает только схему DataTable
RejectChanges()	Выполняется откат всех изменений, внесенных в таблицу с момента её загрузки или после последнего вызова метода AcceptChanges()
Select()	Возвращает массив строк таблицы на основании заданного критерия поиска
WriteXml()	Записывает содержимое DataTable в XML-формате в файл, Stream, TextWriter или XmlWriter
WriteXmlSchema()	Работает как WriteXml(), но записывает только схему таблицы

Класс DataTable может генерировать события, указанные в табл. 7.3.

Таблица 7.3

События класса DataTable

Имя события	Описание
ColumnChanged	Происходит после изменения значения указанного столбца в DataRow
ColumnChanging	Происходит при изменении значения указанного столбца в DataRow
RowChanged	Происходит после успешного изменения DataRow
RowChanging	Происходит при изменении объекта DataRow
RowDeleted	Происходит после удаления строки таблицы
RowDeleting	Происходит перед удалением строки таблицы
TableCleared	Происходит после очистки DataTable
TableClearing	Происходит, когда очищается таблица DataTable
TableNewRow	Происходит, когда вставляется новая строка DataRow

Вспомогательный класс DataTableExtensions определяет для DataTable два метода расширения:

– AsDataView() – возвращает объект DataView с поддержкой LINQ;
– AsEnumerable() – возвращает объект IEnumerable<DataRow>.

Перейдём к рассмотрению класса DataSet. Как и DataTable, DataSet – сериализуемый тип, который реализует интерфейс IListSource. Свойства класса DataSet приведены в табл. 7.4.

Таблица 7.4

Свойства класса DataSet

Имя свойства	Описание
CaseSensitive	Определяет, учитывается ли регистр при сравнении строк в таблицах DataSet
DataSetName	Строка с именем набора данных
DefaultViewManager	Получает новое представление данных класса DataSet для фильтрации или поиска с помощью класса DataViewManager
EnforceConstraints	Определяет, обеспечивает ли DataSet выполнение определённых на нём ограничений
ExtendedProperties	Коллекция пользовательских свойств набора данных
HasErrors	Указывает, содержит ли набор данных ошибки
Locale	Свойство имеет тип CultureInfo и определяет региональные параметры, используемые набором данных при сравнении строк
Namespace	Пространство имён XML класса DataSet
Prefix	Префикс пространства имён XML класса DataSet
Relations	Коллекция отношений, связывающих таблицы из набора данных
RemotingFormat	Позволяет указать формат данных при сериализации DataSet – бинарный или XML
Tables	Возвращает коллекцию таблиц набора данных

Большинство свойств DataSet является аналогами свойств DataTable, но для всего набора данных. В табл. 7.5 перечислены методы DataSet.

Таблица 7.5

Методы класса DataSet

Имя метода	Описание
AcceptChanges()	Метод фиксирует все изменения данных, которые были проделаны с момента предыдущего вызова AcceptChanges()
Clear()	Уничтожаются все строки всех таблиц набора данных
Clone()	Метод клонирует структуру набора и возвращает пустой набор
Copy()	Метод клонирует и структуру, и данные набора
CreateDataReader()	Возвращает объект DataTableReader с одним результирующим набором для каждой таблицы в DataSet
GetChanges()	Возвращает новый DataSet с идентичной схемой, содержащий изменённые строки и таблицы оригинального объекта DataSet
GetXml()	Возвращает содержимое объекта DataSet в виде XML-строки
GetXmlSchema()	Возвращает схему объекта DataSet в виде XML-строки
HasChanges()	Возвращает логическое значение, указывающее, содержат ли строки из состава DataSet отложенные изменения
InferXmlSchema()	Применяет к DataSet XML-схему из указанного файла, Stream, TextReader или XmlReader
IsBinarySerialized()	Анализирует формат сериализованного представления набора данных
Load()	Заполняет DataSet значениями из источника данных с помощью предоставляемого объекта IDataReader
Merge()	Осуществляет слияние данных из другого объекта DataSet, DataTable или массива объектов DataRow и данных текущего объекта DataSet
ReadXml()	Читает содержимое DataSet в XML-формате из файла, Stream, TextReader или XmlReader
ReadXmlSchema()	Работает как ReadXml(), но читает только схему DataSet
RejectChanges()	Метод отменяет изменения, которые еще не зафиксированы вызовом AcceptChanges()
Reset()	Восстанавливает оригинальное состояние DataSet
WriteXml()	Записывает содержимое DataSet в XML-формате в файл, Stream, TextWriter или XmlWriter
WriteXmlSchema()	Работает как WriteXml(), но записывает только схему DataSet

У класса DataSet определено событие MergeFailed, которое происходит при ошибках слияния наборов данных.

8. СХЕМА ТАБЛИЦЫ И КЛАСС DATACOLUMN

Введём понятие схемы рассоединённого набора данных DataSet. Будем считать, что это совокупность следующих элементов:

– имена таблиц;
– типы и имена отдельных столбцов таблицы;

– ограничения на столбцы таблицы: уникальность, отсутствие пустых значений, первичные и внешние ключи;

– связи между таблицами.

Правильная схема обеспечивает контроль целостности данных в приложении. Схема может быть определена вручную, путём создания и настройки свойств столбцов, таблиц, связей. Некоторые (но, к сожалению, не все) элементы схемы создаются автоматически при загрузке данных из базы в пустой `DataSet`. Кроме этого, схемы можно импортировать в виде XSD-файлов.

Структура любой таблицы описывается свойствами её столбцов. Столбец таблицы представлен объектом класса `DataColumn`. Этот класс содержит следующий набор свойств, перечисленных в табл. 8.1.

Таблица 8.1

Свойства класса `DataColumn`

Имя свойства	Тип	Описание
`AllowDBNull`	`bool`	Задаёт, возможны ли в столбце пустые значения
`AutoIncrement`	`bool`	Показывает, генерируется ли для столбца новое значение автоприращения
`AutoIncrementSeed`	`int`	Начальное значение автоприращения
`AutoIncrementStep`	`Int`	Шаг автоприращения
`Caption`	`string`	Заголовок столбца, отображаемый в элементах управления
`ColumnMapping`	`MappingType`	Определяет, как будет записано содержимое столбца при экспорте данных в XML-формате
`ColumnName`	`string`	Имя столбца в таблице
`DataType`	`Type`	Тип данных столбца
`DefaultValue`	`object`	Значение по умолчанию в столбце
`Expression`	`string`	Выражение для *вычисляемых столбцов*
`MaxLength`	`int`	Максимальная длина строковых данных в столбце
`Namespace`	`string`	Пространство имён XML, используемое при экспорте и импорте данных столбца в XML-формате
`Ordinal`	`int`	Порядковый номер столбца в таблице
`Prefix`	`string`	Префикс пространства имён XML
`ReadOnly`	`bool`	Указывает, что содержимое столбца доступно только для чтения
`Table`	`DataTable`	Таблица, в состав которой входит столбец
`Unique`	`bool`	Должно ли быть значение в столбце уникальным в пределах таблицы

Минимально допустимая настройка столбца заключается в указании его имени (`ColumnName`) и типа данных (`DataType`). Класс `DataColumn` имеет конструктор, получающий соответствующие параметры. Можно указать значение по умолчанию в столбце – свойство `DefaultValue`. Если свойство `AllowDBNull` установлено в `true`, то допустимы пустые значения столбца, представленные объектами `System.DBNull`.

При добавлении столбца в таблицу автоматически заполняются его свойства `Ordinal` и `Table`. Метод столбца `SetOrdinal()` позволяет изменить порядковый номер столбца в таблице.

Свойства `AutoIncrement`, `AutoIncrementSeed` и `AutoIncrementStep` используются для организации автоматического приращения значений столбца (по умолчанию автоприращение не активно). Тип свойства с автоприращением должен быть целочисленным. Автоприращение может быть полезно при организации первичного ключа таблицы.

При работе с рассоединённым набором данных существует возможность записать его в формате XML. Свойство `ColumnMapping` настраивает представление столбца при сохранении. Например, если это свойство установлено в `MappingType.Attribute`, значения столбца записываются как XML-атрибуты. К свойствам для поддержки работы с XML относятся также `Namespace` и `Prefix`.

Для хранения столбцов класс `DataTable` использует свойство `Columns` типа `DataColumnCollection`. Добавлять столбцы можно по одному (метод коллекции `Add()`) или целым массивом (метод `AddRange()`). Кроме этого, метод `Add()` имеет удобную перегруженную версию, которая позволяет неявно создать столбец, указав его имя и тип. Пример кода демонстрирует создание схемы для таблицы `Book` (информация о книге) и таблицы `Category` (категория книги).

```
var book = new DataTable("Book");
var bookId = new DataColumn("ID", typeof (int));

var title = new DataColumn("Title", typeof (string));
title.AllowDBNull = false;
title.MaxLength = 500;

var description = new DataColumn("Description", typeof (string));

var isRare = new DataColumn("IsRare", typeof (bool));
isRare.AllowDBNull = false;

var categoryId = new DataColumn("CategoryID", typeof (Guid));
categoryId.AllowDBNull = false;

book.Columns.AddRange(new[] {bookId, title, description,
                             isRare, categoryId});

var category = new DataTable("Category");
category.Columns.Add("ID", typeof (Guid));
category.Columns.Add("Name", typeof (string));
category.Columns["Name"].AllowDBNull = false;
category.Columns["Name"].MaxLength = 50;
```

Заметим, что если схема таблицы формируется при загрузке данных из базы, у столбца автоматически настраиваются следующие свойства: `AllowDBNull`, `Caption`, `ColumnName`, `DataType`, `Ordinal`, `Table`, `Unique`.

Как и в реляционных базах данных, один или несколько столбцов таблицы DataTable могут исполнять роль первичного ключа. Первичный ключ должен быть уникальным в пределах таблицы. Свойство таблицы PrimaryKey служит для получения или установки массива столбцов, формирующих первичный ключ. Если столбец является частью первичного ключа, его свойство AllowDBNull автоматически устанавливается в false. Определим первичные ключи таблиц Book и Category:

```
book.PrimaryKey = new[] { book.Columns["ID"]};
category.PrimaryKey = new[] {category.Columns["ID"]};
```

Таблица DataTable поддерживает свойство Constraints – набор ограничений таблицы. Значением этого свойства является коллекция объектов, унаследованных от Constraint. Абстрактный класс Constraint определяет свойство для имени ограничения (ConstraintName) и для ссылки на таблицу DataTable, содержащую ограничение (Table). У класса Constraint имеются два потомка: UniqueConstraint и ForeignKeyConstraint.

При помощи объектов класса UniqueConstraint описывается уникальность значений в столбце таблицы. Основными свойствами UniqueConstraint являются массив столбцов Columns и булево свойство IsPrimaryKey, которое показывает, представляет ли данное ограничение первичный ключ. Ограничение вида UniqueConstraint автоматически добавляется в таблицу при создании первичного ключа. Также ограничение добавляется, если в таблице есть столбец, у которого свойство Unique установлено в значение true. Допустимо самостоятельно определять объекты UniqueConstraint.

Класс ForeignKeyConstraint служит для описания внешних ключей таблицы. Обычно нет необходимости создавать объекты этого класса вручную. Они генерируются автоматически при добавлении связи между таблицами. Собственные свойства класса ForeignKeyConstraint перечислены в табл. 8.2.

Таблица 8.2

Основные свойства класса ForeignKeyConstraint

Имя свойства	Описание
AcceptRejectRule	Определяет каскадирование вызовов методов AcceptChanges() и RejectChanges() родительского DataRow в дочерние строки
Columns	Столбцы дочерней таблицы, составляющие ограничение
DeleteRule	Определяет каскадирование удаления родительского DataRow в дочерние строки
RelatedColumns	Столбцы родительской таблицы, составляющие ограничение
RelatedTable	Родительская таблица ограничения
UpdateRule	Управляет каскадированием изменений родительской строки в дочерние строки

Свойства AcceptRejectRule, DeleteRule и UpdateRule управляют порядком каскадирования изменений родительской строки в дочерние строки. Свойство AcceptRejectRule принимает значения из одноименного перечисления.

По умолчанию используется AcceptRejectRule.None – вызов AcceptChanges() или RejectChanges() у объекта DataRow родительской таблицы не сказывается на дочерних строках. Если указать значение AcceptRejectRule.Cascade, изменения каскадируются в дочерние строки. Свойства DeleteRule и UpdateRule функционируют аналогичным образом, но принимают значения из перечисления Rule. Значение этих свойств по умолчанию – Rule.Cascade, то есть изменения родительской строки каскадируются в дочерние строки. Например, при вызове метода Delete() у родительского объекта DataRow неявно вызывается метод Delete() его дочерних строк. Если каскадировать изменения не требуется, нужно задать свойствам DeleteRule и UpdateRule значение Rule.None. Можно также использовать значения Rule.SetNull или Rule.SetDefault. В первом случае при изменении или удалении содержимого родительской строки соответствующим полям дочерних строк задаются значения NULL, а во втором – их значения по умолчанию.

9. СХЕМА РАССОЕДИНЁННОГО НАБОРА ДАННЫХ

Создадим схему рассоединённого набора данных, взяв за основу схему таблиц Book и Category. Для этого сконструируем объект DataSet, а затем добавим таблицы в его коллекцию Tables:

```
var dataSet = new DataSet("Library");
dataSet.Tables.Add(book);
dataSet.Tables.Add(category);
```

Добавим связь между таблицами Book и Category. Связь представлена объектом класса DataRelation. В табл. 9.1 перечислены свойства этого класса.

Таблица 9.1

Свойства класса DataRelation

Имя свойства	Описание
ChildColumns	Свойство возвращает массив, содержащий объекты DataColumn из дочернего объекта DataTable
ChildKeyConstraint	Связанное ограничение ForeignKeyConstraint в дочерней таблице
ChildTable	Указывает дочернюю таблицу связи
DataSet	Набор данных, в котором находится объект DataRelation
ExtendedProperties	Набор динамических свойств
Nested	Логическое значение. Указывает, нужно ли преобразовывать дочерние строки в дочерние элементы при записи содержимого DataSet в XML-файл
ParentColumns	Родительские столбцы, определяющие отношение

Окончание табл. 9.1

Имя свойства	Описание
`ParentKeyConstraint`	Связанное ограничение `UniqueConstraint` в родительской таблице
`ParentTable`	Указывает родительскую таблицу связи
`RelationName`	Строка с именем отношения

При создании объекта `DataRelation` следует указать его имя, чтобы объект удалось найти в наборе; кроме этого, необходимо указать родительские и дочерние столбцы, на которых основано отношение. Чтобы упростить создание связей, класс `DataRelation` предоставляет конструкторы, принимающие как отдельные объекты `DataColumn`, так и массивы таких объектов.

Вернемся к нашему примеру. Между таблицами `Book` и `Category` существует связь по внешнему ключу. А именно, таблица `Book` является дочерней для таблицы `Category`, так как значения колонки с именем `CategoryID` в `Book` – это значения первичного ключа таблицы `Category`. Создадим объект `DataRelation`, описывающий эту связь:

```
// используем конструктор, который устанавливает имя отношения,
// а также родительский и дочерний столбцы отношения
var categoryToBook = new DataRelation("CategoryToBook",
                                      category.Columns["ID"],
                                      book.Columns["CategoryID"]);
dataSet.Relations.Add(categoryToBook);
```

На этом создание схемы для набора данных можно считать завершённым. При помощи метода набора `WriteXmlSchema()` схему можно сохранить в XSD-файле:

```
dataSet.WriteXmlSchema("Library.xsd");
```

Располагая схемой в виде XSD-файла, можно сгенерировать типизированный набор данных. *Типизированный набор данных* – это совокупность классов, унаследованных от `DataSet`, `DataTable`, `DataRow`. В типизированном наборе для доступа к отдельным компонентам используются свойства, а не индексаторы по имени элемента. Это устанавливает контроль над правильностью кода во время компиляции. Следующий фрагмент демонстрирует использование обычного и типизированного набора данных:

```
// обычный набор данных
var s1 = (string) dataSet.Tables["Category"].Rows[0]["Name"];

// типизированный набор данных
var library = new Library();
string s2 = library.Category[0].Name;
```

Для создания типизированного набора на основе XSD-файла применяется утилита `xsd.exe`, входящая в состав .NET Framework SDK:

```
xsd.exe Library.xsd /dataset
```

Результатом работы утилиты `xsd.exe` является файл `Library.cs`, который содержит определения классов `Library` (наследник `DataSet`), `BookDataTable` и `CategoryDataTable` (оба – наследники `DataTable`), `BookRow` и `CategoryRow` (наследники `DataRow`), `BookRowChangeEvent` и `CategoryRowChangeEvent` (наследники `EventArgs`).

В Visual Studio 2010 типизированный набор данных можно создать в проекте при помощи команд `Project` ▯ `Add New Item` ▯ `Data` ▯ `DataSet` ▯ *`задать имя xsd-файла`*. В результате в проект добавляются файлы, описывающие типизированный набор данных, и появляется возможность визуального редактирования набора (рис. 9.1).

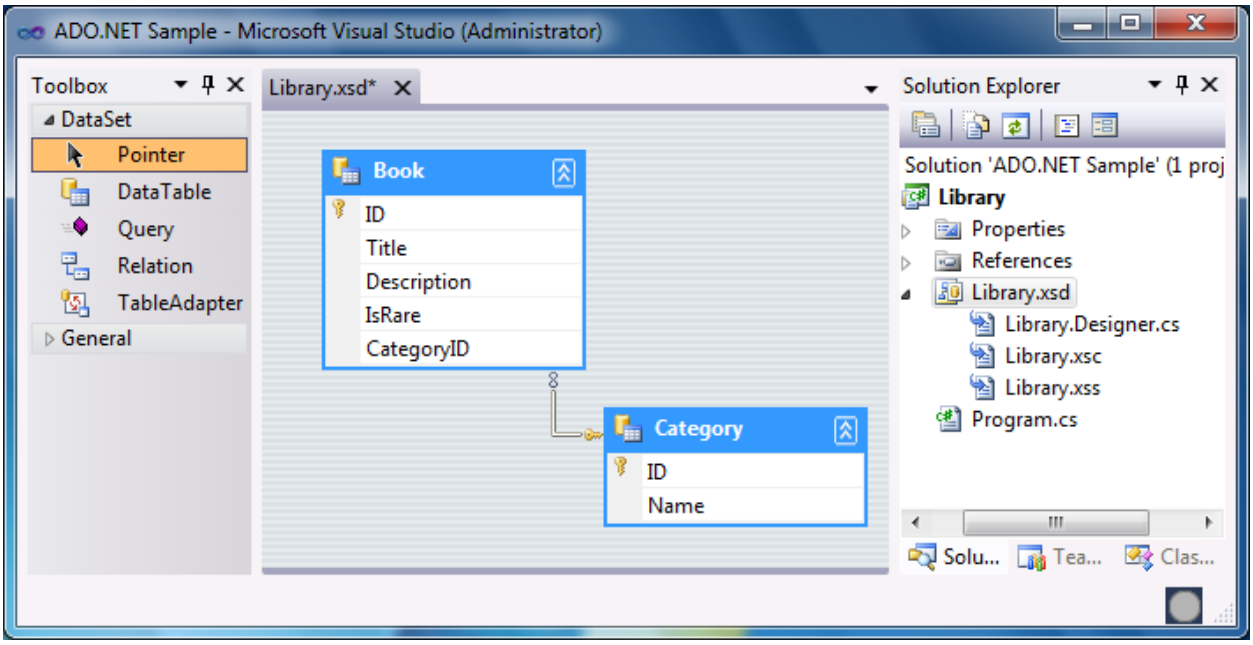

Рис. 9.1. Редактирование типизированного `DataSet` в Visual Studio

10. ЗАПОЛНЕНИЕ РАССОЕДИНЁННОГО НАБОРА ДАННЫХ

Каждый поставщик данных содержит класс, описывающий адаптер данных. Адаптер является своеобразным мостом между источником данных и рассоединённым набором. Он позволяет считывать информацию из источника в набор и производит обратную операцию. В принципе подобные действия вполне осуществимы при помощи команд и объектов *`DataReader`*. Использование адаптера данных – это более унифицированный подход.

Любой адаптер реализует интерфейс `IDataAdapter` и наследуется от класса `DataAdapter`. Основные элементы `DataAdapter` описаны в табл. 10.1.

Таблица 10.1

Основные элементы класса DataAdapter

Имя элемента	Описание
AcceptChangesDuringFill	Свойство, которое указывает, вызывается ли метод AcceptChanges() в объекте DataRow после его добавления к таблице при выполнении метода Fill()
AcceptChangesDuringUpdate	Свойство указывает, вызывается ли метод AcceptChanges() при выполнении метода Update()
ContinueUpdateOnError	Свойство указывает, следует ли генерировать исключение при ошибке во время обновления строки
Fill()	Метод добавляет или обновляет строки в DataSet или таблице для получения соответствия строкам в источнике данных
FillError	Это событие генерируется при возникновении ошибки во время работы операции заполнения
FillLoadOption	Значение из перечисления LoadOption. Определяет, как адаптер заполняет DataTable из объекта *DataReader*
FillSchema()	Добавляет таблицу к указанному объекту DataSet и настраивает схему таблицы
GetFillParameters()	Получает параметры, заданные пользователем при выполнении оператора SELECT
MissingMappingAction	Определяет действие, выполняемое, если входные данные не соответствуют таблице или столбцу
MissingSchemaAction	Определяет действие, выполняемое, если существующая схема DataSet не соответствует входным данным
ResetFillLoadOption()	Сбрасывает свойство FillLoadOption к состоянию по умолчанию
ReturnProviderSpecificTypes	Свойство указывает, должен ли метод Fill() возвращать зависящие от поставщика значения или обычные CLS-совместимые значения
TableMappings	Коллекция, обеспечивающая основное сопоставление между исходной таблицей и объектом DataTable
Update()	Вызывает операторы INSERT, UPDATE или DELETE для каждой вставленной, обновлённой или удалённой строки в указанном наборе данных или таблице

Рассмотрим работу с классом SqlDataAdapter, описанным в поставщике данных для SQL Server. SqlDataAdapter дополняет класс DataAdapter свойствами SelectCommand, InsertCommand, DeleteCommand и UpdateCommand. Это объекты SqlCommand для выборки и обновления источника данных. Перед началом работы с адаптером его нужно создать и инициализировать свойства-команды (пока будет использоваться только команда для выборки данных). Адаптер содержит несколько перегруженных конструкторов. Варианты вызова конструкторов показаны в примере кода:

```
// обычный конструктор без параметров
var adapter1 = new SqlDataAdapter();

// команда выборки данных передаётся как параметр конструктора
var command = new SqlCommand("SELECT * FROM Book");
var adapter2 = new SqlDataAdapter(command);

// параметры: текст команды выборки данных и объект соединения
var connect = new SqlConnection(builder.ConnectionString);
var adapter3 = new SqlDataAdapter("SELECT * FROM Book", connect);

// параметры: текст запроса и строка подключения
var adapter4 = new SqlDataAdapter("SELECT * FROM Book",
                                  builder.ConnectionString);
```

После создания адаптера можно использовать его метод `Fill()` для заполнения набора данных `DataSet` или таблицы `DataTable`:

```
var dataSet = new DataSet();
adapter3.Fill(dataSet);

var dataTable = new DataTable();
adapter4.Fill(dataTable);
```

Заметим, что вызов метода `Fill()` не нарушает состояние соединения с источником данных для команды выборки. Если соединение было открыто до вызова `Fill()`, оно останется открытым и после вызова. Если соединение было не установлено, метод `Fill()` откроет соединение, произведёт выборку данных и закроет соединение. Так же ведут себя и остальные методы адаптера, работающие с источником данных.

Поведение адаптера при заполнении `DataSet` зависит от настроек адаптера и наличия схемы в объекте `DataSet`. Если при помощи адаптера заполняется пустой рассоединённый набор, то адаптер строит в `DataSet` *минимальную схему*, используя имена и тип столбцов из источника данных и стандартные имена для таблиц. В результате выполнения следующего кода в `dataSet` будет создана одна таблица с именем `Table`:

```
var adapter = new SqlDataAdapter("SELECT * FROM Book",
                                 builder.ConnectionString);
var dataSet = new DataSet();
adapter.Fill(dataSet);
```

Команда выборки данных может быть настроена на получение нескольких наборов данных. В следующем примере в пустой `DataSet` помещаются две таблицы с именами `Table` и `Table1`:

```
var text = "SELECT * FROM Book; SELECT * FROM Category";
var adapter = new SqlDataAdapter(text, builder.ConnectionString);
```

```
var dataSet = new DataSet();
adapter.Fill(dataSet);
```

Адаптер имеет свойство-коллекцию TableMappings, которое позволяет сопоставить имена получаемых наборов данных и таблиц DataSet. Любой элемент коллекции TableMappings содержит коллекцию ColumnMappings, которая осуществляет отображение имён столбцов:

```
string text = "SELECT * FROM Book; SELECT * FROM Category";
var adapter = new SqlDataAdapter(text, builder.ConnectionString);

DataTableMapping mapping;
mapping = adapter.TableMappings.Add("Table", "Book");
mapping.ColumnMappings.Add("Title", "Caption");
```

Предположим, что заполняемый DataSet уже обладает некоторой схемой. Адаптер содержит свойство MissingSchemaAction, значениями которого являются элементы одноимённого перечисления. По умолчанию свойство имеет значение MissingSchemaAction.Add. Это означает добавление в схему новых элементов, если они в ней не описаны. Возможными значениями являются также MissingSchemaAction.Ignore (игнорирование элементов, не известных схеме) и MissingSchemaAction.Error (если элементы не описаны в схеме, генерируется исключение).

Обсудим дополнительные возможности адаптера, связанные с заполнением DataSet. Адаптер имеет метод FillSchema(), который переносит схему таблиц запроса в DataSet. Метод FillSchema() получает из источника данных имена и типы всех задействованных в запросе столбцов. Кроме этого, данный метод получает сведения о допустимости для столбца значений NULL и задаёт корректные значения свойств AllowDBNull создаваемых им объектов DataColumn. Метод FillSchema() также пытается определить на объекте DataTable первичный ключ.

Для заполнения таблицы информацией может использоваться отдельная группа методов класса DataTable, не связанная с адаптером данных. Метод Load() формирует содержимое строк таблицы с помощью объекта IDataReader, передаваемого как параметр. Метод LoadDataRow() добавляет (или обновляет) строку таблицы на основе данных массива, принимаемого в качестве параметра. Элементы массива должны соответствовать элементам коллекции Columns таблицы. Для импорта строки из сторонней таблицы подходит метод ImportRow().

11. ХРАНЕНИЕ ДАННЫХ В СТРОКЕ И КЛАСС DATAROW

Таблица DataTable содержит коллекцию Rows для хранения строк с данными. Отдельная строка представлена объектом класса DataRow. Свойства класса DataRow перечислены в табл. 11.1, а набор методов – в табл. 11.2.

Таблица 11.1

Свойства класса DataRow

Имя свойства	Описание
HasErrors	Указывает, есть ли ошибки в строке
Item[]	Перегруженные индексаторы для доступа к полям строки. Поддерживают обращение по индексу и имени поля, а также с использованием объекта DataColumn
ItemArray	Возвращает или задаёт все значения строки с помощью массива
RowError	Строковое описание ошибки для строки
RowState	Текущее состояние строки
Table	Ссылка на таблицу DataTable, владеющую строкой

Таблица 11.2

Методы класса DataRow

Имя метода	Описание
AcceptChanges()	Фиксирует все изменения, внесённые в эту строку со времени последнего вызова метода AcceptChanges()
BeginEdit()	Начинает операцию изменения объекта DataRow
CancelEdit()	Отменяет текущее изменение строки
ClearErrors()	Очищает ошибки для строки. Это относится к свойству RowError и ошибкам, установленным с помощью метода SetColumnError()
Delete()	Удаляет объект DataRow
EndEdit()	Прекращает изменение строки
GetChildRows()	Получает дочерние строки объекта DataRow с помощью отношения между таблицами
GetColumnError()	Получает описание ошибки для поля, указанного как параметр
GetColumnsInError()	Возвращает массив столбцов с ошибками
GetParentRow()	Получает родительскую строку объекта DataRow с помощью отношения между таблицами
HasVersion()	Получает значение, показывающее, существует ли указанная версия строки
IsNull()	Проверяет, содержит ли указанное поле строки значение NULL
RejectChanges()	Отменяет все изменения, внесённые в строку после последнего вызова метода AcceptChanges()
SetAdded()	Изменяет значение свойства RowState на RowState.Added
SetColumnError()	Задаёт строку с описанием ошибки для поля, указанного как параметр
SetModified()	Изменяет значение свойства RowState на RowState.Modified
SetNull()	Задаёт значение NULL для поля, указанного как параметр
SetParentRow()	Задаёт родительскую строку DataRow с указанным новым родительским объектом DataRow

Рассмотрим некоторые приёмы работы с объектом DataRow (будем использовать в примерах таблицу category, сконструированную ранее и храня-

щую категории книг). Чтобы отредактировать или прочитать данные строки, требуется получить эту строку из таблицы. В простейшем случае для этого используется номер строки в табличной коллекции `Rows`[1]:

```
// получаем вторую строку из таблицы category
DataRow row = category.Rows[1];
```

Класс `DataRow` предоставляет несколько индексаторов для доступа к полям строки. В качестве индекса может использоваться имя столбца, номер столбца или объект `DataColumn`, представляющий столбец:

```
row["ID"] = Guid.NewGuid();
string name = (string) row[1];
```

Класс `DataRowExtensions` содержит два универсальных метода расширения для класса `DataRow`: `Field<T>()` и `SetField<T>()`. Эти методы осуществляют типизированный доступ к полям строки. Как и индексаторы класса `DataRow`, методы `Field<T>()` и `SetField<T>()` имеют различные перегруженные версии:

```
row.SetField("ID", Guid.NewGuid());
var name = row.Field<string>(1);
```

Свойство строки `ItemArray` позволяет просматривать и редактировать содержимое всей строки. Его тип – массив объектов, элементы которого соответствуют полям строки. Если необходимо отредактировать содержимое лишь некоторых полей, следует для неизменяемых полей использовать `null`:

```
// меняем содержимое второго поля строки
row.ItemArray = new object[] {null, "Mystery"};
```

Возможен способ редактирования строки с буферизацией изменений. В этом случае перед началом редактирования нужно вызвать у строки метод `BeginEdit()`. При вызове в конце редактирования `EndEdit()` коррективы сохраняются в строке. Если нужно отменить их, следует использовать метод `CancelEdit()`, и строка вернется в состояние на момент вызова `BeginEdit()`.

```
row.BeginEdit();
row.SetField("ID", Guid.NewGuid());
row.EndEdit();
```

Объект `DataTable` предоставляет события `RowChanging`, `RowChanged`, `ColumnChanging` и `ColumnChanged`, с помощью которых удаётся отслеживать изменения строки или поля. Порядок наступления этих событий зависит от того, как редактируется строка – с вызовом методов `BeginEdit()` и `EndEdit()` или без них. Если вызван метод `BeginEdit()`, наступление событий откладывается до вызова `EndEdit()` (если вызвать `CancelEdit()` никакие события не наступают).

[1] О других способах получения и поиска строк таблицы будет рассказано ниже

Если нужно создать для выбранной таблицы новую строку, следует использовать метод таблицы `NewRow()`. Он генерирует пустую строку по схеме таблицы, но не добавляет эту строку в таблицу. Для добавления строки необходимо заполнить её данными, а затем воспользоваться методом `Add()` табличной коллекции `Rows`. Существует перегруженный вариант метода `Add()`, принимающий в качестве параметра массив объектов, являющихся значениями полей строки:

```
// 1 - создали пустую строку с требуемой структурой
DataRow row = category.NewRow();

// 2 - заполняем поля строки
row["ID"] = Guid.NewGuid();
row["Name"] = "Horror";

// 3 - добавляем в таблицу
category.Rows.Add(row);

// короткий вариант, в котором совмещены сразу три действия
category.Rows.Add(new object[] {Guid.NewGuid(), "Humor"});
```

Чтобы удалить строку, достаточно вызвать метод `Delete()` объекта `DataRow`. После вызова `Delete()` строка помечается как удаленная; в источнике данных она будет фактически удалена при синхронизации источника и рассоединённого набора. Можно удалить строку из коллекции `Rows` таблицы, воспользовавшись методами коллекции `Remove()` или `RemoveAt()`. Однако, если строка удалена подобным образом, то строка не удаляется из источника данных при синхронизации.

Пусть имеется некий рассоединённый набор данных, в который помещена информация из источника данных. Допустим, что информация была изменена (в таблицах редактировались, добавлялись или удалялись строки), и необходимо переместить содержимое набора обратно в источник данных. Для эффективной синхронизации следует применить *отслеживание изменений* в наборе данных и внесение в источник только корректирующих поправок.

Для поддержки корректирующих изменений каждая строка имеет *состояние* и *версию*. Состояние хранится в свойстве строки `RowState` и принимает значения из перечисления `DataRowState`:

– `Unchanged` – строка не менялась (совпадает со строкой в источнике данных);

– `Detached` – строка не относится к объекту `DataTable`;

– `Added` – строка добавлена в объект `DataTable`, но не существует в источнике данных;

– `Modified` – строка была изменена по сравнению со строкой в источнике данных;

– `Deleted` – строка ожидает удаления из источника данных.

В табл. 11.3 показано, как может изменяться состояние отдельной строки.

Таблица 11.3

Изменение состояния строки DataRow

Действие	Пример кода	Значение RowState
Создание новой строки, не добавленной к объекту DataTable	`row = tbl.NewRow();` `row["ID"] = 100;`	Detached
Добавление строки в DataTable	`tbl.Rows.Add(row);`	Added
Получение существующей строки	`row = tbl.Rows[0];`	Unchanged

Редактирование строки	`row.BeginEdit();`
	`row["ID"] = 2000;`
	`row.EndEdit();`
		Modified

Удаление строки	`row.Delete();`	Deleted

С помощью методов SetAdded() и SetModified() свойство строки RowState может быть изменено программно. Использование этих методов приводит к принудительной передаче данных в источник данных, даже если сами данные не претерпели изменений.

Кроме использования RowState для любой строки существует возможность просмотреть, каким было значение её полей *до изменения*. Индексатор строки и методы расширения Field<T>() и SetField<T>() имеют перегруженные версии, принимающие значения из перечисления DataRowVersion:

– Current – текущее значение поля;
– Original – оригинальное значение поля;
– Proposed – предполагаемое значение поля (действительно только при редактировании строки с использованием BeginEdit()).

Следующий фрагмент кода изменяет поле, а затем выводит оригинальное и текущее содержимое поля:

```
// заполняем таблицу из базы, чтобы была оригинальная версия строки
var book = new DataTable("Book");
var adapter = new SqlDataAdapter("SELECT * FROM Book", connection);
adapter.Fill(book);

// меняем содержимое поля первой строки
var row = book.Rows[0];
row["Title"] = "The new book";

// выводим различные версии поля
Console.WriteLine(row["Title", DataRowVersion.Current]);
Console.WriteLine(row["Title", DataRowVersion.Original]);
```

В табл. 11.4 показаны возможные значения, возвращаемые индексатором в зависимости от указанной версии ([Искл.] обозначает генерацию исключительной ситуации при попытке получить определенную версию).

Таблица 11.4

Значения индексатора в зависимости от версии строки

Пример	Current	Original	Proposed

Строка создана, но не связана с таблицей
`row = tbl.NewRow();`
`row["ID"] = 10;`
	[Искл.]	[Искл.]	10

В таблицу добавлена новая строка `tbl.Rows.Add(row);`	10	[Искл.]	[Искл.]
Данные загружены из источника данных, из таблицы получена существующая строка `row = tbl.Rows[0];`	1[1]	1	[Искл.]

Первое изменение существующего поля
`row.BeginEdit();`
`row["ID"] = 100;`
	1	1	100

После первого изменения `row.EndEdit();`	100	1	[Искл.]

После второго изменения содержимого поля
`row.BeginEdit();`
`row["ID"] = 300;`
`row.EndEdit();`
	300	1	[Искл.]

После отмены изменений
`row.BeginEdit();`
`row["ID"] = 500;`
`row.CancelEdit()`
	300	1	[Искл.]

После удаления записи
`DataRow row = tbl.Rows[0];`
`row.Delete();`
	[Искл.]	1	[Искл.]

Для контроля существования версии можно использовать метод строки `HasVersion()`:

```
if (row.HasVersion(DataRowVersion.Current))
{
    Console.WriteLine("Current version exists");
}
```

Любая строка предоставляет методы `AcceptChanges()` и `RejectChanges()`. Вызов `AcceptChanges()` приводит к замене значений `Original`-версии значениями из `Current`-версии и установке у строки свойства `RowState` в `DataRowState.Unchanged`. Вызов `RejectChanges()` также устанавливает `RowState` в `DataRowState.Unchanged`, но значения `Current`-версии строки меняются на значения `Original`-версии. При загрузке изменений `DataSet` в источник данных у каждой строки неявно вызывается метод `AcceptChanges()`. Следует знать, что явное использование указанных методов может породить проблемы при синхронизации набора данных.

[1] Это значение получено из базы

12. ВЫБОРКА ДАННЫХ ИЗ DATATABLE

Создадим и заполним информацией таблицу `person`, которая будет использоваться в дальнейших примерах:

```
// класс хранит информацию о человеке
public class Person
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}

// создадим коллекцию объектов
var list = new List<Person>
           {
               new Person {ID = 1, Name = "George", Age = 57},
               new Person {ID = 2, Name = "John", Age = 61},
               new Person {ID = 3, Name = "Thomas", Age = 57},
               new Person {ID = 4, Name = "James", Age = 57}
           };

// сконструируем схему таблицы
var person = new DataTable("Person");
person.Columns.Add("ID", typeof (int));
person.Columns.Add("Name", typeof (string));
person.Columns.Add("Age", typeof (int));
person.PrimaryKey = new[] {person.Columns["ID"]};

// заполним таблицу информацией из коллекции
foreach (Person p in list)
{
    person.LoadDataRow(new object[] {p.ID, p.Name, p.Age}, true);
}
```

Для поиска и выборки информации из таблицы, в которой определён первичный ключ, может использоваться метод `Find()`, который предоставляет класс `DataRowCollection`. Метод `Find()` принимает как параметр значение первичного ключа искомой строки. Поскольку значения первичного ключа уникальны, метод вернёт не более одного объекта `DataRow` (или `null`, если информация не найдена):

```
DataRow row = person.Rows.Find(1);
if (row != null)
{
    Console.WriteLine(row["Name"]);
}
```

Метод Find() перегружен в расчёте на случаи, когда первичный ключ DataTable состоит из нескольких объектов DataColumn. В этом случае в Find() передаётся массив объектов, представляющих значения первичного ключа.

Метод Select() класса DataTable позволяет искать строки по определённому критерию, который описывается строкой и передаётся как параметр метода. Метод Select() возвращает массив найденных строк:

```
DataRow[] rows = person.Select("Age > 60");
foreach (DataRow row in rows)
{
    Console.WriteLine(row["Name"]);
}
```

Метод Select() перегружен и позволяет указать не только критерий поиска, но и настроить порядок сортировки найденных строк. Также можно искать строки с указанным значением состояния и версии.

```
// выборка и сортировка по полю Name
DataRow[] rows = person.Select("Age > 50", "Name");
```

Объект класса DataTable не является перечисляемым. Однако класс DataTableExtensions определяет для DataTable метод расширения AsEnumerable(), который представляет таблицу как набор IEnumerable<DataRow>. Это даёт возможность применить для выборки и обработки данных операторы LINQ to Objects:

```
var names = from p in person.AsEnumerable()
            where p.Field<int>(2) > 60
            select p.Field<string>("Name");
foreach (string name in names)
{
    Console.WriteLine(name);
}
```

Такие операции LINQ to Objects как Distinct(), Union(), Intersect() и Except() по умолчанию используют семантику на основе ссылок для выяснения равенства объектов. При применении этих операций к набору строк таблицы следует использовать экземпляр DataRowComparer.Default, чтобы обеспечить правильную семантику сравнения объектов DataRow.

13. КЛАСС DATAVIEW

В ADO.NET имеется специальный класс DataView для представления результатов фильтрации, сортировки и поиска в таблице DataTable. Объекты DataView не являются SQL-запросами, в отличие от *представлений* (*view*) в базах данных. С помощью DataView нельзя объединить данные двух таблиц, равно как и просмотреть отдельные столбцы таблицы. Объекты DataView поддерживают фильтрацию запросов на основе динамических критериев, но разрешают

обращаться только к одному объекту DataTable. Кроме этого, через DataView всегда доступны все столбцы таблицы. У объекта DataView нет собственной копии данных. При обращении через DataView к данным он возвращает строки, хранящиеся в соответствующем объекте DataTable.

Чтобы просмотреть с помощью объекта DataView данные некоторой таблицы, этот объект следует связать с таблицей. Для этого можно использовать конструктор DataView или свойство Table объекта DataView. Кроме этого, у объекта DataTable есть свойство DefaultView, которое указывает на представление по умолчанию для таблицы.

```
// три способа получения объекта DataView
var dataView = new DataView(person);

var dataView2 = new DataView();
dataView2.Table = person;

var dataView3 = person.DefaultView;
```

Три свойства DataView управляют представлением информации: RowFilter, RowStateFilter и Sort. Строковое свойство RowFilter описывает критерий фильтрации (аналогично параметру метода Select() класса DataTable). Свойство Sort задаёт строку с именем столбца (или столбцов), по которым выполняется сортировка данных. Свойство RowStateFilter принимает значения из перечисления DataViewRowState. Это перечисление можно рассматривать как комбинацию свойства RowState объекта DataRow и перечисления DataRowVersion:

– Added – добавленные строки;
– CurrentRows – строки, которые не были удалены (по умолчанию);
– Deleted – удалённые строки;
– ModifiedCurrent – изменённые строки с их текущими значениями;
– ModifiedOriginal – изменённые строки с их оригинальными значениями;
– None – строки не отображаются;
– OriginalRows – удалённые, изменённые и не изменившиеся строки с их оригинальными значениями;
– Unchanged – строки, которые не изменялись.

Свойство RowStateFilter работает в качестве двойного фильтра. Например, если задать ему значение DataViewRowState.ModifiedOriginal, через объект DataView окажутся видны только изменённые строки и будут доступны только оригинальные значения их полей.

```
var dataView = new DataView(person);
dataView.RowFilter = "Age > 50";
dataView.Sort = "Name DESC";
dataView.RowStateFilter = DataViewRowState.Unchanged;
```

Определён конструктор класса DataView, который позволяет задать связанную таблицу и значения свойств RowFilter, Sort, RowStateFilter:

```
var dataView = new DataView(person, "Age > 50", "Name DESC",
                            DataViewRowState.CurrentRows);
```

Объект DataView хранит строки представления с помощью специализированного объекта DataRowView. Возможности DataRowView в целом аналогичны функциям DataRow, но через DataRowView доступна только одна версия данных – та, которая указана в представлении при помощи свойства RowStateFilter. Класс DataRowView предоставляет индексатор, позволяющий обратится к содержимому поля по имени или по номеру. Если объект DataRowView не обеспечивает требуемых возможностей, при помощи свойства Row этого объекта можно получить соответствующий объект DataRow из таблицы.

Чтобы перебрать все строки представления, можно использовать целочисленный индексатор и свойство Count, возвращающее число строк. Кроме этого, класс DataView реализует интерфейс IEnumerable.

```
var dataView = person.DefaultView;
foreach (DataRowView row in dataView)
{
    Console.WriteLine(row["Name"]);
}
```

Класс DataView предоставляет методы Find() и FindRows(), позволяющие искать данные по указанному ключу сортировки (по значениям столбцов, перечисленных в свойстве Sort). Метод Find() принимает одно значение или массив объектов, а возвращает число, соответствующее порядковому номеру первой найденной строки в объекте DataView. Метод FindRows() возвращает массив объектов DataRowView – строки, удовлетворяющие критериям поиска.

```
var dataView = person.DefaultView;
dataView.Sort = "Age";

int index = dataView.Find(61);
if (index != -1)
{
    Console.WriteLine(dataView[index]["Name"]);
}

DataRowView[] rows = dataView.FindRows(57);
foreach (DataRowView r in rows)
{
    Console.WriteLine(r["Name"]);
}
```

Так как DataView реализует интерфейс IEnumerable, для поиска строк можно использовать LINQ to Objects (предварительно вызвав у объекта DataView метод Cast<DataRowView>()):

```
var dataView = person.DefaultView;
var rows = dataView.Cast<DataRowView>()
                   .Where(r => (int)r["Age"] > 57);
foreach (DataRowView dataRowView in rows)
{
    Console.WriteLine(dataRowView["Name"]);
}
```

Строка данных модифицируется с помощью объекта DataRowView аналогично изменению содержимого объекта DataRow. Создание новой строки при помощи объекта DataRowView несколько отличается от создания нового объекта DataRow. У класса DataView есть метод AddNew(), возвращающий новый объект DataRowView. В действительности же новая строка добавляется в базовый объект DataTable только при вызове метода EndEdit() объекта DataRowView. Ниже показано, как средствами объекта DataRowView можно создать, изменить и удалить строку данных:

```
var dataView = person.DefaultView;

// создание новой строки
DataRowView row = dataView.AddNew();
row["ID"] = 5;
row["Name"] = "James";
row["Age"] = 60;
row.EndEdit();

// получение и редактирование строки
row = dataView[4];
row.BeginEdit();
row["Age"] = 58;
row.EndEdit();

// получение и удаление строки
row = dataView[4];
row.Delete();
```

В заключение заметим, что класс DataView имеет метод ToTable(), позволяющий сформировать таблицу на основе представления. Различные перегрузки метода ToTable() пригодны для управления именем создаваемой таблицы, столбцами таблицы, уникальностью строк таблицы.

```
// создаём и настраиваем представление
var dataView = person.DefaultView;
dataView.Sort = "Age";
```

```
// создаём таблицу President со столбцами ID, Name
// без поддержки уникальности строк (второй параметр)
var table = dataView.ToTable("President", false, "ID", "Name");
```

14. СИНХРОНИЗАЦИЯ НАБОРА И ИСТОЧНИКА ДАННЫХ

Пусть рассоединённый набор заполняется информацией из источника данных при помощи адаптера:

```
var builder = new SqlConnectionStringBuilder();
builder.DataSource = @"(local)\SQLExpress";
builder.InitialCatalog = "Library";
builder.IntegratedSecurity = true;
var connection = new SqlConnection(builder.ConnectionString);

var dataSet = new DataSet("Library");
var adapter = new SqlDataAdapter("SELECT * FROM Category",
                                 connection);
adapter.Fill(dataSet, "Category");
```

Если набор был изменён и данные требуется синхронизировать с источником, следует использовать метод адаптера `Update()`. Параметрами метода `Update()` могут быть таблица, массив объектов `DataRow` или объект `DataSet`. Однако попытка выполнения следующего кода вызовет исключительную ситуацию `InvalidOperationException`:

```
dataSet.Tables["Category"].Rows[0]["Name"] = "Empty";
adapter.Update(dataSet, "Category");
// System.InvalidOperationException: Update requires a valid
// UpdateCommand when passed DataRow collection with modified rows
```

Дело в том, что при создании адаптера, используемого в примере, была автоматически сформирована и помещена в свойство `SelectCommand` только одна команда – команда для выборки данных. Остальные свойства-команды адаптера не инициализированы.

Программист может настроить необходимые команды вручную. SQL-синтаксис возможных команд для вставки, обновления и удаления данных в таблице `Category` приведён ниже:

```
INSERT INTO Category(ID, Name) VALUES (@p1, @p2)

DELETE FROM Category WHERE (ID = @p1) AND (Name = @p2)

UPDATE Category SET ID = @p1, Name = @p2
                WHERE (ID = @p3) AND (Name = @p4)
```

Эти SQL-инструкции следует скопировать в свойство `CommandText` соответствующих команд. Отдельного пояснения требует настройка параметров. Параметр кроме установки таких свойств, как имя и тип, должен быть связан со

столбцом таблицы из набора данных, а в случае с командой UPDATE – еще и с определенной версией информации в столбце. Для этого используются свойства параметра SourceColumn и SourceVersion. Приведем полный текст создания и настройки команд вставки, обновления и удаления:

```
// создаём три объекта-команды
SqlCommand insert = connection.CreateCommand();
SqlCommand delete = connection.CreateCommand();
SqlCommand update = connection.CreateCommand();

// настраиваем текст команд
insert.CommandText =
          "INSERT INTO Category(ID, Name) VALUES (@p1, @p2)";
delete.CommandText =
          "DELETE FROM Category WHERE (ID = @p1) AND (Name = @p2)";
update.CommandText = "UPDATE Category SET ID = @p1, Name = @p2 " +
                     "WHERE (ID = @p3) AND (Name = @p4)";

// создадим два параметра для insert и поместим их в коллекцию
insert.Parameters.Add("@p1", SqlDbType.UniqueIdentifier);
insert.Parameters.Add("@p2", SqlDbType.VarChar);

// дополнительная настройка – укажем столбец, из которого
// берётся значение параметра
insert.Parameters[0].SourceColumn = "ID";
insert.Parameters[1].SourceColumn = "Name";

// в случае с командой удаления – аналогичные действия
delete.Parameters.Add("@p1", SqlDbType.UniqueIdentifier);
delete.Parameters.Add("@p2", SqlDbType.VarChar);
delete.Parameters[0].SourceColumn = "ID";
delete.Parameters[1].SourceColumn = "Name";

// для команды обновления число параметров в два раза больше
update.Parameters.Add("@p1", SqlDbType.UniqueIdentifier);
update.Parameters.Add("@p2", SqlDbType.VarChar);
update.Parameters.Add("@p3", SqlDbType.UniqueIdentifier);
update.Parameters.Add("@p4", SqlDbType.VarChar);
update.Parameters[0].SourceColumn = "ID";
update.Parameters[1].SourceColumn = "Name";
update.Parameters[2].SourceColumn = "ID";
update.Parameters[3].SourceColumn = "Name";

// требуется указать версию поля таблицы
update.Parameters[2].SourceVersion = DataRowVersion.Original;
update.Parameters[3].SourceVersion = DataRowVersion.Original;

// помещаем команды в адаптер
adapter.InsertCommand = insert;
```

```
adapter.DeleteCommand = delete;
adapter.UpdateCommand = update;
```

После того как в адаптере определены все команды, можно свободно изменять данные в рассоединенном наборе, а затем обновить их в источнике вызовом метода адаптера `Update()`.

Как показывает пример, ручное создание команд для адаптера даже в случае простого набора данных выглядит громоздким (хотя это достаточно гибкое решение). Поставщики данных предоставляют построитель команд *`CommandBuilder`* для автоматической генерации команд адаптера. Для использования построителя команд нужно создать объект *`CommandBuilder`* и связать его с адаптером данных, у которого уже задана команда `SelectCommand`. После установки подобной связи *`CommandBuilder`* отслеживает событие обновления строки данных, которое происходит при вызове метода `Update()`, и автоматически генерирует необходимые команды на основе текста команды выборки.

Приведём пример кода для синхронизации, использующий построитель команд `SqlCommandBuilder`, описанный в поставщике для SQL Server.

```
// создаём объект CommandBuilder и связываем его с адаптером
var builder = new SqlCommandBuilder(adapter);

// после этого можно вызывать метод адаптера Update()
adapter.Update(dataSet, "Category");
```

Отметим, что объект *`CommandBuilder`* может сгенерировать правильные команды обновления, если выполняются следующие три условия:

– запрос возвращает данные только из одной таблицы;
– на таблице в источнике данных определён первичный ключ;
– в наборе столбцов результата запроса присутствует первичный ключ или уникальный столбец.

Анализ команд, генерируемых объектом `SqlCommandBuilder`, показывает, что данный класс ориентирован на использование стратегии *оптимистического параллелизма* (*optimistic concurrency*) при возникновении конфликтов обновления данных.

ЛИТЕРАТУРА

1. Гамильтон, Б. ADO.NET Сборник рецептов. Для профессионалов / Б. Гамильтон ; пер. с англ. – СПб. : Питер, 2005. – 576 с.

2. Малик, С. Microsoft ADO.NET 2.0 для профессионалов / С. Малик ; пер. с англ. – М. : ООО «Изд. Дом Вильямс», 2006. – 560 с.

3. Сеппа, Д. Программирование на Microsoft® ADO.NET 2.0. Мастер-класс / Д. Сеппа ; пер. с англ. – М. : Изд. «Русская редакция» ; СПб. : Питер, 2007. – 784 с.

4. Троелсен, Э. Язык программирования C# 2010 и платформа .NET 4.0 / Э. Троелсен ; пер. с англ. – М. : ООО «И. Д. Вильямс», 2010. – 1392 с.

Св. план 2011, поз. 53

Учебное издание

Волосевич Алексей Александрович
Актанорович Сергей Владимирович

СРЕДСТВА ПЛАТФОРМЫ .NET

ДЛЯ РАБОТЫ С БАЗАМИ ДАННЫХ

Методическое пособие
по дисциплинам «Инструменты и средства программирования»
и «Избранные главы информатики»
для студентов специальности 1-31 03 04 «Информатика»
всех форм обучения

Редактор Н. В. Гриневич
Корректор А. В. Тюхай
Компьютерная верстка Ю. Ч. Клочкевич

Подписано в печать 16.02.2011. Формат 60x84 1/16. Бумага офсетная.
Гарнитура «Таймс». Отпечатано на ризографе. Усл. печ. л. 3,14.
Уч.-изд. л. 3,1. Тираж 120 экз. Заказ 815.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0494371 от 16.03.2009. ЛП №02330/0494175 от 03.04.2009.
220013, Минск, П. Бровки, 6