ПРИЛОЖЕНИЕ A

ADO.NET: наборы данных, таблицы данных и адаптеры данных

В главе 21 исследовались подключенный уровень и фундаментальные компоненты ADO.NET, которые позволяют отправлять в базу данных операторы SQL с использованием объекта подключения, объектов команд и объекта чтения данных выбранного поставщика данных. В настоящем приложении вы ознакомитесь с *автономным уровнем* ADO.NET. Эта грань ADO.NET позволяет моделировать данные из базы в памяти, внутри вызывающего уровня, за счет применения многочисленных членов из пространства имен `System.Data` (в особенности `DataSet`, `DataTable`, `DataRow`, `DataColumn`, `DataView` и `DataRelation`). В таком случае можно обеспечить иллюзию того, что вызывающий уровень постоянно подключен к внешнему источнику данных, хотя в реальности все операции происходят с локальной копией реляционных данных.

В приложении A также будут демонстрироваться некоторые приемы привязки данных с использованием контекста настольного приложения, имеющего графический пользовательский интерфейс Windows Forms, и раскрыта роль *строго типизированного* объекта `DataSet`. Мы обновим созданную в главе 21 библиотеку доступа к данным `AutoLotDAL.dll`, добавив новое пространство имен, в котором задействован автономный уровень ADO.NET. В завершение приложения мы обсудим технологию LINQ to DataSet, которая позволяет применять запросы LINQ к находящемуся в памяти кешу данных.

Понятие автономного уровня ADO.NET

Подключенный уровень позволяет взаимодействовать с базой данных с использованием исходных объектов подключения, команд и чтения данных. Имеющийся небольшой набор классов можно применять для выборки, вставки, обновления и удаления записей из основного содержимого (а также вызывать хранимые процедуры или выполнять другие операции над данными (например, DDL для создания таблицы и DCL для выдачи разрешений)). Однако это только часть истории ADO.NET. Вспомните, что объектную модель ADO.NET можно использовать также в автономной манере.

С применением автономного уровня появляется возможность моделирования реляционных данных с помощью объектной модели, расположенной в памяти. Выходя далеко за пределы простого моделирования табличных блоков строк и столбцов, типы из пространства имен `System.Data` позволяют представлять отношения между таблица-

ми, ограничения столбцов, первичные ключи, представления и другие примитивы баз данных. После построения модели данных к ней можно применять фильтры, отправлять запросы в памяти и сохранять (или загружать) данные в формате XML и в двоичном формате. Все перечисленное можно делать, даже не подключаясь к СУБД (отсюда и название *автономный уровень*), а загружая данные из локального файла XML или вручную создавая объект `DataSet` в коде.

Автономные типы можно было бы использовать, не подключаясь к базе данных, но обычно вы по-прежнему будете применять объекты подключений и команд. Вдобавок для выборки и обновления данных будет задействован специфичный объект — *адаптер данных* (тип которого расширяет абстрактный класс `DbDataAdapter`). В отличие от подключенного уровня данные, полученные посредством адаптера данных, не обрабатываются с использованием объектов чтения данных. Взамен объекты адаптеров перемещают данные между вызывающим кодом и источником данных с применением объектов `DataSet` (точнее объектов `DataTable` в `DataSet`). Тип `DataSet` представляет собой контейнер для любого количества объектов `DataTable`, каждый из которых содержит коллекцию объектов `DataRow` и `DataColumn`.

Объект адаптера данных вашего поставщика данных автоматически обслуживает подключение к базе данных. В стремлении увеличить масштабируемость адаптеры данных удерживают подключение открытым в течение минимально возможного времени. После получения объекта `DataSet` вызываемый уровень полностью отключается от базы данных и остается с локальной копией удаленных данных. Вызывающий код может вставлять, удалять или модифицировать строки в выбранном объекте `DataTable`, но физическая база данных не обновляется до тех пор, пока в вызывающем коде не будет явно передан объект `DataTable` из `DataSet` адаптеру данных для обновления. По существу объекты `DataSet` позволяют клиентам делать вид, что они постоянно подключены, хотя фактически они оперируют с базой данных, находящейся в памяти (рис. А.1).

Рис. А.1. Объекты адаптеров данных перемещают объекты `DataSet` на клиентский уровень и обратно

Поскольку центральной частью автономного уровня является класс `DataSet`, то первой задачей приложения будет объяснение способа манипулирования им вручную. Освоив такое манипулирование, никаких проблем с обработкой содержимого `DataSet`, извлеченного из объекта адаптера данных, у вас не возникнет.

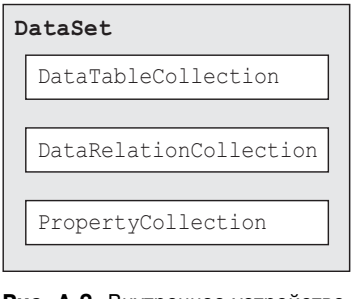

Рис. А.2. Внутреннее устройство класса `DataSet`

Роль объектов `DataSet`

Как отмечалось ранее, объект `DataSet` представляет реляционные данные в памяти. Более конкретно `DataSet` — это класс, который внутренне поддерживает три строго типизированных коллекции (рис. А.2).

Свойство `Tables` класса `DataSet` предоставляет доступ к коллекции `DataTableCollection`, которая

содержит отдельные объекты DataTable. В DataSet присутствует еще одна важная коллекция — DataRelationCollection.

Учитывая, что объект DataSet является автономной версией схемы базы данных, его можно использовать для программного представления отношений "родительский–дочерний" между таблицами. Например, с применением типа DataRelation можно создать отношение между двумя таблицами для моделирования ограничения внешнего ключа. Затем результирующий объект DataRelation можно добавить в коллекцию DataRelation Collection через свойство Relations. В результате при поиске данных можно перемещаться между связанными таблицами. Вы увидите, как это делать, далее в приложении.

Свойство ExtendedProperties предоставляет доступ к объекту PropertyCollection, который позволяет ассоциировать с DataSet любую дополнительную информацию в виде пар "имя-значение". Информация может быть совершенно произвольной, даже не имеющей отношения к самим данным. Например, с объектом DataSet можно связать название компании, которое затем будет выступать в качестве метаданных в памяти. Другими примерами расширенных свойств могут служить отметки времени, зашифрованный пароль, который должен быть указан для доступа к содержимому DataSet, число, представляющее частоту обновления данных, и многое другое.

На заметку! Классы DataTable и DataColumn также поддерживают свойство Extended Properties.

Основные свойства класса DataSet

Прежде чем погрузиться во множество других деталей программирования, давайте взглянем на ряд основных членов класса DataSet. Помимо свойств Tables, Relations и ExtendedProperties класс DataSet определяет дополнительные полезные свойства, которые кратко описаны в табл. A.1.

Таблица A.1. Свойства класса DataSet

Свойство	Описание
CaseSensitive	Указывает, чувствительно ли к регистру сравнение строк в объектах DataTable. По умолчанию равно false (сравнение строк выполняется без учета регистра)
DataSetName	Представляет дружественное имя объекта DataSet. Обычно это значение передается в параметре конструктора
EnforceConstraints	Получает или устанавливает значение, которое определяет, соблюдаются ли правила ограничений при попытке выполнить любые операции обновления (по умолчанию равно true)
HasErrors	Получает значение, указывающее на то, есть ли ошибки в любой строке любого объекта DataTable внутри DataSet
RemotingFormat	Позволяет определить, каким образом объект DataSet должен сериализировать свое содержимое (в двоичном виде или по умолчанию в XML)

Основные методы класса DataSet

Методы класса DataSet работают в сочетании с функциональностью, предоставляемой упомянутыми выше свойствами. В дополнение к взаимодействию с потоками XML класс DataSet предлагает методы, позволяющие копировать содержимое DataSet, перемещаться между внутренними таблицами и устанавливать начальные и конечные точки пакета обновлений. В табл. A.2 описаны избранные основные методы.

Таблица A.2. Избранные основные методы класса `DataSet`

Метод	Описание
`AcceptChanges()`	Фиксирует все изменения, внесенные в текущий объект `DataSet` с момента его загрузки или последнего вызова метода `AcceptChanges()`
`Clear()`	Полностью очищает данные `DataSet`, удаляя все строки из каждого объекта `DataTable`
`Clone()`	Клонирует структуру, но не данные `DataSet`, включая все объекты `DataTable`, а также все отношения и любые ограничения
`Copy()`	Копирует структуру и данные текущего объекта `DataSet`
`GetChanges()`	Возвращает копию объекта `DataSet`, которая содержит все изменения, внесенные в объект `DataSet` с момента его загрузки или последнего вызова `AcceptChanges()`. Этот метод имеет перегруженные версии, позволяющие получать только новые строки, только измененные строки или только удаленные строки
`HasChanges()`	Получает значение, которое указывает, был ли объект `DataSet` изменен, включая добавление, удаление либо изменение строк
`Merge()`	Объединяет текущий объект `DataSet` с указанным объектом `DataSet`
`ReadXml()`	Позволяет определить структуру объекта `DataSet` и заполнить его данными, основываясь на XML-схеме и данных из потока
`RejectChanges()`	Производит откат всех изменений, которые были внесены в текущий объект `DataSet` с момента его загрузки или последнего вызова метода `AcceptChanges()`
`WriteXml()`	Позволяет записать содержимое объекта `DataSet` в действительный поток

Построение объекта `DataSet`

Теперь, когда вы лучше понимаете роль класса `DataSet` (и имеете некоторое представление о том, что можно делать с его помощью), создадим новый проект консольного приложения по имени `SimpleDataSet` и импортируем пространство имен `System.Data`. Внутри метода `Main()` определим новый объект `DataSet`, содержащий три расширенных свойства, которые представляют отметку времени, уникальный идентификатор (типа `System.Guid`) и название компании (также понадобится добавить оператор `using static System.Console;`):

```
using static System.Console;
static void Main(string[] args)
{
  WriteLine("***** Fun with DataSets *****\n");

  // Создать объект DataSet и добавить несколько свойств.
  var carsInventoryDS = new DataSet("Car Inventory");

  carsInventoryDS.ExtendedProperties["TimeStamp"] = DateTime.Now;
  carsInventoryDS.ExtendedProperties["DataSetID"] = Guid.NewGuid();
  carsInventoryDS.ExtendedProperties["Company"] =
    "Mikko's Hot Tub Super Store";

  FillDataSet(carsInventoryDS);
  PrintDataSet(carsInventoryDS);

  ReadLine();
}
```

На заметку! Глобально уникальный идентификатор (globally unique identifier — GUID) представляет собой статически уникальное 128-битное число.

Объект `DataSet` не особенно интересен, пока не вставить в него несколько объектов `DataTable`. Следовательно, понадобится исследовать внутреннее устройство `DataTable`, начав с типа `DataColumn`.

Работа с объектами `DataColumn`

Тип `DataColumn` представляет одиночный столбец внутри объекта `DataTable`. В целом набор всех объектов `DataColumn`, привязанных к заданному объекту `DataTable`, представляет основу информации *схемы* таблицы. Например, в случае моделирования таблицы `Inventory` из базы данных `AutoLot` (см. главу 21) будут созданы четыре объекта `DataColumn`, по одному для каждого столбца (`CarId`, `Make`, `Color` и `PetName`). Полученные объекты `DataColumn` обычно добавляются в коллекцию столбцов объекта `DataTable` (с использованием свойства `Columns`).

Вероятно, вы уже знаете, что столбцу в таблице базы данных можно назначить набор ограничений (например, сконфигурировать как первичный ключ, присвоить стандартное значение или разрешить только чтение). Кроме того, каждый столбец таблицы должен отображаться на лежащий в основе тип данных. Скажем, схема таблицы `Inventory` требует, чтобы столбец `CarId` отображался на целочисленное значение, а столбцы `Make`, `Color` и `PetName` — на массив символов. Класс `DataColumn` имеет множество свойств, которые позволяют точно конфигурировать такие аспекты. Описание ряда основных свойств приведено в табл. A.3.

Таблица A.3. Свойства класса `DataColumn`

Свойство	Описание
`AllowDBNull`	Применяется для указания, может ли столбец содержать значения `null`. По умолчанию равно `true`

`AutoIncrement`
`AutoIncrementSeed`
`AutoIncrementStep`
	Используются в целях настройки поведения автоинкремента для заданного столбца. Это может быть удобно, когда нужно гарантировать уникальность значений в отдельном объекте `DataColumn` (таком как первичный ключ). По умолчанию `DataColumn` не поддерживает поведение автоинкремента

`Caption`	Получает или устанавливает заголовок, который необходимо отображать для столбца. Это позволяет определить дружественную к пользователю версию для реального имени столбца в базе данных
`ColumnMapping`	Определяет представление объекта `DataColumn` при сохранении `DataSet` в документе XML посредством метода `DataSet.WriteXml()`. Можно указать, что столбец данных должен быть записан как элемент XML, атрибут XML, простое текстовое содержимое или вообще игнорироваться
`ColumnName`	Получает или устанавливает имя столбца в коллекции `Columns` (т.е. как он представлен внутри `DataTable`). Если свойство `ColumnName` не установлено явно, то стандартным значением будет слово `Column` с числовым суффиксом по формуле *n*+1 (`Column1`, `Column2`, `Column3` и т.д.)
`DataType`	Определяет тип (например, булевский, строковый или с плавающей точкой) данных, хранящихся в столбце

Окончание табл. A.3

Свойство	Описание
DefaultValue	Получает или устанавливает стандартное значение, присваиваемое столбцу при вставке новых строк
Expression	Получает или устанавливает выражение для фильтрации строк, вычисления значения столбца или создания агрегированного столбца
Ordinal	Получает или устанавливает числовую позицию столбца в коллекции Columns, поддерживаемой объектом DataTable
ReadOnly	Определяет, допускает ли заданный столбец только чтение, после добавления строки в таблицу. Стандартным значением является false
Table	Получает объект DataTable, который содержит текущий объект DataColumn
Unique	Получает или устанавливает значение, которое указывает, должны ли быть значения во всех строках столбца уникальными, или же разрешены повторяющиеся значения. Когда столбцу назначается ограничение первичного ключа, свойство Unique должно быть установлено в true

Построение объекта DataColumn

Чтобы продолжить работу с проектом SimpleDataSet (и проиллюстрировать применение типа DataColumn), предположим, что необходимо смоделировать столбцы таблицы Inventory. Поскольку столбец CarId будет первичным ключом таблицы, он должен быть сконфигурирован как предназначенный только для чтения, содержащий уникальные значения и не допускающий null (с использованием свойств ReadOnly, Unique и AllowDBNull). Добавим в класс Program новый метод по имени FillDataSet(), который будет применяться для построения четырех объектов DataColumn. Метод FillDataSet() принимает в качестве единственного параметра объект DataSet.

```
static void FillDataSet(DataSet ds)
{
  // Создать столбцы данных, которые отображаются на "реальные"
  // столбцы в таблице Inventory из базы данных AutoLot.
  var carIDColumn = new DataColumn("CarID", typeof (int))
  {
    Caption = "Car ID",
    ReadOnly = true,
    AllowDBNull = false,
    Unique = true,
  };
  var carMakeColumn = new DataColumn("Make", typeof (string));
  var carColorColumn = new DataColumn("Color", typeof (string));
  var carPetNameColumn = new DataColumn("PetName", typeof (string))
    { Caption = "Pet Name"};
}
```

Обратите внимание, что при конфигурировании объекта carIDColumn свойству Caption было присвоено значение. Это свойство удобно, т.к. позволяет определить строковое значение в целях отображения, которое может отличаться от реального имени столбца таблицы в базе данных (имена столбцов обычно больше подходят для целей программирования (например, au_fname), чем для отображения (скажем, Author First Name (Имя автора))). По той же причине был установлен заголовок для столбца PetName, потому что Pet Name (Дружественное имя) выглядит для конечного пользователя лучше, чем PetName.

Включение автоинкрементных полей

Одним из аспектов типа DataColumn, который вы можете выбрать для конфигурирования, является возможность *автоинкремента*. Автоинкрементное поле используется для обеспечения того, что при добавлении в таблицу новой строки значение такого поля устанавливается автоматически на основе предыдущего значения и шага увеличения. Это удобно, когда нужно гарантировать, что в столбце отсутствуют повторяющиеся значения (например, в первичном ключе).

Указанное поведение управляется свойствами AutoIncrement, AutoIncrementSeed и AutoIncrementStep. Свойство AutoIncrementSeed применяется для определения начального значения в столбце, а свойство AutoIncrementStep позволяет задать число, которое прибавляется при вычислении каждого последующего значения. Взгляните на следующее обновление кода создания объекта carIDColumn:

```
static void FillDataSet(DataSet ds)
{
  var carIDColumn = new DataColumn("CarID", typeof (int))
  {
    Caption = "Car ID",
    ReadOnly = true,
    AllowDBNull = false,
    Unique = true,
    AutoIncrement = true,
    AutoIncrementSeed = 1,
    AutoIncrementStep = 1
  };
}
```

Здесь объект carIDColumn сконфигурирован так, что при добавлении новых строк в соответствующую таблицу значение в столбце увеличивается на 1. Начальное значение установлено в 1, поэтому в столбце будут содержаться числа 1, 2, 3, 4 и т.д.

Добавление объектов DataColumn в DataTable

Обычно объект типа DataColumn не существует как обособленная сущность, а вставляется в связанный объект DataTable. Для примера создадим новый объект DataTable и вставим все объекты DataColumn в коллекцию столбцов с использованием свойства Columns:

```
static void FillDataSet(DataSet ds):
{
  ...
  // Добавить объекты DataColumn в DataTable.
  var inventoryTable = new DataTable("Inventory");
  inventoryTable.Columns.AddRange(new[]
    {carIDColumn, carMakeColumn, carColorColumn, carPetNameColumn});
}
```

В настоящий момент объект DataTable содержит четыре объекта DataColumn, которые представляют схему находящейся в памяти таблицы Inventory. Тем не менее, пока что таблица Inventory не содержит данных и не входит в коллекцию таблиц, обслуживаемых DataSet. Мы восполним эти пробелы, начав с наполнения таблицы данными с применением объектов DataRow.

Работа с объектами DataRow

Вы видели, что коллекция объектов DataColumn представляет схему DataTable. С другой стороны, коллекция объектов DataRow представляет действительные данные в таблице. Таким образом, если таблица Inventory базы данных AutoLot содержит 20 строк, то представить их можно с использованием 20 объектов DataRow.

В табл. А.4 кратко описаны некоторые (но не все) члены типа DataRow.

Таблица А.4. Основные члены типа DataRow

Член	Описание
HasErrors GetColumnsInError() GetColumnError() ClearErrors() RowError	Свойство HasErrors возвращает булевское значение, указывающее на наличие ошибок в DataRow. Если ошибки есть, то с помощью метода GetColumnsInError() можно получить проблемные столбцы, а с помощью метода GetColumnError() — описание ошибки. Метод ClearErrors() позволяет очистить список ошибок для строки. Свойство RowError дает возможность сконфигурировать текстовое описание ошибки для заданной строки
ItemArray	Это свойство получает или устанавливает значения всех столбцов в строке с применением массива объектов
RowState	Это свойство используется для определения текущего *состояния* объекта DataRow в содержащем его объекте DataTable посредством значений перечисления RowState (например, строка может быть помечена как новая, модифицированная, неизмененная или удаленная)
Table	Это свойство применяется для получения ссылки на объект DataTable, содержащий данный объект DataRow
AcceptChanges() RejectChanges()	Эти методы фиксируют или отклоняют все изменения, внесенные в данную строку с момента последнего вызова AcceptChanges()
BeginEdit() EndEdit() CancelEdit()	Эти методы начинают, заканчивают или отменяют операцию редактирования для объекта DataRow
Delete()	Этот метод помечает строку, которую необходимо удалить при вызове метода AcceptChanges()
IsNull()	Этот метод возвращает значение, которое указывает, содержит ли заданный столбец null

Работа с объектом DataRow слегка отличается от работы с DataColumn; создавать экземпляр типа DataRow напрямую невозможно, т.к. открытые конструкторы в нем отсутствуют:

```
// Ошибка! Открытых конструкторов нет!
DataRow r = new DataRow();
```

Взамен новый объект DataRow получается из заданного объекта DataTable. Пусть в таблицу Inventory нужно вставить две строки. Метод DataTable.NewRow() позволяет получить очередную область в таблице, после чего каждый столбец можно заполнить новыми данными, используя индексатор типа. При этом можно указывать либо строковое имя, назначенное объекту DataColumn, либо порядковый номер (начинающийся с нуля):

```
static void FillDataSet(DataSet ds)
{
  ...
  // Добавить несколько строк в таблицу Inventory.
  DataRow carRow = inventoryTable.NewRow();
  carRow["Make"] = "BMW";
  carRow["Color"] = "Black";
  carRow["PetName"] = "Hamlet";
  inventoryTable.Rows.Add(carRow);

  carRow = inventoryTable.NewRow();
  // Столбец 0 - автоинкрементное поле идентификатора,
  // поэтому начать заполнение со столбца 1.
  carRow[1] = "Saab";
  carRow[2] = "Red";
  carRow[3] = "Sea Breeze";
  inventoryTable.Rows.Add(carRow);
}
```

На заметку! Если методу индексатора типа `DataRow` передается недействительное имя столбца или порядковый номер позиции, то во время выполнения генерируется исключение.

В настоящий момент имеется единственный объект `DataTable`, содержащий две строки. Разумеется, такой общий процесс можно повторить, чтобы определить схему и содержимое для нескольких объектов `DataTable`. Перед вставкой объекта `inventoryTable` в `DataSet` вы должны ознакомиться с крайне важным свойством `RowState`.

Свойство `RowState`

Свойство `RowState` применяется для программной идентификации набора строк таблицы, которые были изменены, заново вставлены и т.д. Данному свойству можно присваивать любое значение из перечисления `DataRowState` (табл. A.5).

Таблица A.5. Значения перечисления `DataRowState`

Значение	Описание
`Added`	Строка была добавлена в `DataRowCollection`, а метод `AcceptChanges()` пока не был вызван
`Deleted`	Строка была помечена для удаления с помощью метода `Delete()` класса `DataRow`, а метод `AcceptChanges()` пока не был вызван
`Detached`	Строка была создана, но не является частью какой-то коллекции `DataRowCollection`. Объект `DataRow` находится в этом состоянии непосредственно после создания, но перед добавлением в коллекцию. Он также будет в таком состоянии после удаления из коллекции
`Modified`	Строка была изменена, а метод `AcceptChanges()` пока не был вызван
`Unchanged`	Строка не изменялась с момента последнего вызова метода `AcceptChanges()`

При программном манипулировании строками заданного объекта `DataTable` свойство `RowState` устанавливается автоматически. Добавим в класс `Program` новый метод, который работает с локальным объектом `DataRow`, попутно выводя на консоль состояние его строк:

```
private static void ManipulateDataRowState()
{
  // Создать объект temp типа DataTable для целей тестирования.
  var temp = new DataTable("Temp");
  temp.Columns.Add(new DataColumn("TempColumn", typeof(int)));
  // RowState = Detached.
  var row = temp.NewRow();
  WriteLine($"After calling NewRow(): {row.RowState}");
  // RowState = Added.
  temp.Rows.Add(row);
  WriteLine($"After calling Rows.Add(): {row.RowState}");
  // RowState = Added.
  row["TempColumn"] = 10;
  WriteLine($"After first assignment: {row.RowState}");
  // RowState = Unchanged.
  temp.AcceptChanges();
  WriteLine($"After calling AcceptChanges: {row.RowState}");
  // RowState = Modified.
  row["TempColumn"] = 11;
  WriteLine($"After first assignment: {row.RowState}");
  // RowState = Deleted.
  temp.Rows[0].Delete();
  WriteLine($"After calling Delete: {row.RowState}");
}
```

На заметку! Не забудьте добавить оператор `using static System.Console;` в начало файла кода в рассматриваемом примере (и во всех других примерах, где используется консоль).

Тип `DataRow` в ADO.NET достаточно интеллектуален, чтобы запоминать свое текущее положение дел. В итоге владеющий им объект `DataTable` способен идентифицировать, какие строки были добавлены, обновлены или удалены. Это важная особенность `DataSet`, т.к. когда наступает время передачи обновленной информации хранилищу данных, отправляются только модифицированные данные.

Свойство `DataRowVersion`

Помимо отслеживания текущего состояния строки посредством свойства `RowState` объект `DataRow` поддерживает три возможных версии содержащихся в нем данных с помощью свойства `DataRowVersion`. Когда объект `DataRow` сконструирован впервые, он содержит только единственную копию данных, которая является текущей версией. Однако по мере программного манипулирования объектом `DataRow` (с применением разнообразных методов) появляются дополнительные версии данных. Свойство `DataRowVersion` может быть установлено в любое значение перечисления `DataRowVersion` (табл. А.6).

Таблица А.6. Значения перечисления `DataRowVersion`

Значение	Описание
`Current`	Представляет текущее значение строки, даже после внесения изменений
`Default`	Стандартная версия `DataRowState`. Если значение `DataRowState` равно `Added`, `Modified` или `Deleted`, то стандартной версией является `Current`. Для значения `DataRowState`, равного `Detached`, стандартной версией будет `Proposed`
`Original`	Представляет значение, первоначально вставленное в `DataRow`, или значение при последнем вызове метода `AcceptChanges()`
`Proposed`	Значение строки, редактируемой в текущий момент по причине вызова `BeginEdit()`

В табл. A.6 видно, что значение свойства DataRowVersion во многих случаях зависит от значения свойства DataRowState. Как упоминалось ранее, свойство DataRowVersion будет автоматически изменяться при вызовах различных методов на объекте DataRow (или в ряде ситуаций на объекте DataTable). Ниже приведен обзор методов, которые могут повлиять на значение свойства DataRowVersion отдельной строки.

- Если вызывается метод DataRow.BeginEdit() и изменяется значение строки, то становятся доступными значения Current и Proposed.
- Если вызывается метод DataRow.CancelEdit(), то значение Proposed удаляется.
- После вызова метода DataRow.EndEdit() значение Proposed становится значением Current.
- После вызова метода DataRow.AcceptChanges() значение Original становится идентичным значению Current. То же самое происходит и при вызове метода DataTable.AcceptChanges().
- После вызова метода DataRow.RejectChanges() значение Proposed отбрасывается, и версия становится Current.

Действительно, ситуация выглядит немного запутанной, и не в последнюю очередь из-за того, что в любое время объект DataRow может иметь или же не иметь все версии (при попытке получить версию строки, которая в текущий момент не отслеживается, возникают исключения времени выполнения). Несмотря на сложность, поскольку объект DataRow поддерживает три копии данных, облегчается построение пользовательского интерфейса, который позволяет конечному пользователю изменить значения и затем отказаться от изменений или зафиксировать новые значения для постоянного сохранения. В оставшемся материале приложения вы увидите разнообразные примеры манипулирования такими методами.

Работа с объектами DataTable

В типе DataTable определены многочисленные члены, многие из которых идентичны по именам и функциональности членам типа DataSet. В табл. A.7 кратко описаны основные члены типа DataTable кроме Rows и Columns.

Таблица A.7. Основные члены типа DataTable

Член	Описание
CaseSensitive	Указывает, должно ли сравнение строк внутри таблицы быть чувствительным к регистру символов. Стандартное значение равно false
ChildRelations	Возвращает коллекцию дочерних отношений для этого объекта DataTable (если они есть)
Constraints	Возвращает коллекцию ограничений, поддерживаемых таблицей
Copy()	Метод, который копирует схему и данные из объекта DataTable в новый такой объект
DataSet	Возвращает объект DataSet, содержащий эту таблицу (если он есть)
DefaultView	Возвращает настроенное представление таблицы, которое может включать отфильтрованное представление или позицию курсора
ParentRelations	Возвращает коллекцию родительских отношений для этого объекта DataTable
PrimaryKey	Получает или устанавливает массив столбцов, которые выступают в качестве первичных ключей для таблицы данных
TableName	Получает или устанавливает имя таблицы. Значение для этого свойства можно также указать в параметре конструктора

Чтобы продолжить текущий пример, присвоим свойству PrimaryKey в DataTable объект DataColumn по имени carIDColumn. Имейте в виду, что для учета ключа, состоящего из нескольких столбцов, свойству PrimaryKey должна присваиваться коллекция объектов DataColumn. Тем не менее, в рассматриваемом случае необходимо указать только столбец CarId (находящийся в самой первой позиции таблицы):

```
static void FillDataSet(DataSet ds)
{
  ...
  // Установить первичный ключ таблицы.
  inventoryTable.PrimaryKey = new [] { inventoryTable.Columns[0] };
}
```

Вставка объектов DataTable в DataSet

К настоящему моменту построение объекта DataTable завершено. Осталось вставить его в объект DataSet по имени carsInventoryDS, используя коллекцию Tables:

```
static void FillDataSet(DataSet ds)
{
  ...
  // Наконец, добавить таблицу в DataSet.
  ds.Tables.Add(inventoryTable);
}
```

Теперь обновим метод Main(), добавив вызов метода FillDataSet(), которому в качестве аргумента передается локальный объект DataSet. Затем передадим этот же объект новому (пока еще не написанному) вспомогательному методу по имени PrintDataSet():

```
static void Main(string[] args)
{
  WriteLine("***** Fun with DataSets *****\n");
  ...
  FillDataSet(carsInventoryDS);
  PrintDataSet(carsInventoryDS);
  ReadLine();
}
```

Получение данных из объекта DataSet

Метод PrintDataSet() просто проходит по метаданным DataSet (с применением коллекции ExtendedProperties) и по каждому объекту DataTable внутри DataSet, выводя на консоль имена столбцов и значения строк с помощью индексаторов типов. Добавим в начало файла оператор using для пространства имен System.Collections, чтобы получить доступ к типу DictionaryEntry:

```
static void PrintDataSet(DataSet ds)
{
  // Вывести имя DataSet и любые расширенные свойства.
  WriteLine($"DataSet is named: {ds.DataSetName}");
  foreach (DictionaryEntry de in ds.ExtendedProperties)
  {
    WriteLine($"Key = {de.Key}, Value = {de.Value}");
  }
  WriteLine();

  // Вывести содержимое каждой таблицы.
  foreach (DataTable dt in ds.Tables)
```

```
  {
    WriteLine($"=> {dt.TableName} Table:");
    // Вывести имена столбцов.
    for (var curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
      Write($"{dt.Columns[curCol].ColumnName}\t");
    }
    WriteLine("\n----------------------------------");
    // Вывести содержимое DataTable.
    for (var curRow = 0; curRow < dt.Rows.Count; curRow++)
    {
      for (var curCol = 0; curCol < dt.Columns.Count; curCol++)
      {
        Write($"{dt.Rows[curRow][curCol]}\t");
      }
      WriteLine();
    }
  }
}
```

Если теперь запустить программу, то будет получен следующий вывод (конечно, метка времени и значение GUID у вас будут другими):

```
***** Fun with DataSets *****

DataSet is named: Car Inventory
Key = TimeStamp, Value = 7/24/2015 6:41:09 AM
Key = DataSetID, Value = 11c533ed-d1aa-4c82-96d4-b0f88893ab21
Key = Company, Value = Mikko's Hot Tub Super Store

=> Inventory Table:
CarID    Make     Color    PetName
---------------------------------
1        BMW      Black    Hamlet
2        Saab     Red      Sea Breeze
```

Обработка данных в `DataTable` с использованием объектов `DataTableReader`

Учитывая проделанную в главе 21 работу, вы должны заметить, что способы обработки данных с применением подключенного уровня (объектов чтения данных) и автономного уровня (объектов `DataSet`) совершенно отличаются. Работа с объектом чтения данных обычно предусматривает организацию цикла `while`, вызов метода `Read()` и выборку пар "имя-значение" с использованием индексатора. С другой стороны, при обработке объекта `DataSet` обычно задействуется последовательность итерационных конструкций для обращения к данным внутри таблиц, строк и столбцов (вспомните, что объект `DataReader` требует открытого подключения к базе данных, чтобы он мог читать данные из действительной базы).

Объекты `DataTable` поддерживают метод по имени `CreateDataReader()`. Он позволяет получать данные из `DataTable` с применением схемы навигации, которая похожа на схему, реализованную объектом чтения данных (теперь данные будут читаться из объекта `DataTable` в памяти, а не из действительной базы данных, так что подключение к базе данных здесь не участвует). Главное преимущество такого подхода связано с тем, что для обработки данных используется единая модель независимо от того, какой уровень ADO.NET применяется для получения данных.

Предположим, что в класс `Program` добавлен новый метод по имени `PrintTable()`:

```
static void PrintTable(DataTable dt)
{
  // Получить объект DataTableReader.
  DataTableReader dtReader = dt.CreateDataReader();

  // DataTableReader работает в точности как DataReader.
  while (dtReader.Read())
  {
    for (var i = 0; i < dtReader.FieldCount; i++)
    {
      Write($"{dtReader.GetValue(i).ToString().Trim()}\t");
    }
    WriteLine();
  }
  dtReader.Close();
}
```

Обратите внимание, что `DataTableReader` работает идентично объекту чтения данных вашего поставщика данных. Тип `DataTableReader` может быть идеальным вариантом, когда нужно быстро извлечь данные из `DataTable` без необходимости в обходе внутренних коллекций строк и столбцов. Далее предположим, что предыдущий метод `PrintDataSet()` модифицирован для вызова `PrintTable()` вместо работы с коллекциями `Rows` и `Columns`:

```
static void PrintDataSet(DataSet ds)
{
  // Вывести имя DataSet и любые расширенные свойства.
  WriteLine($"DataSet is named: {ds.DataSetName}");
  foreach (DictionaryEntry de in ds.ExtendedProperties)
  {
    WriteLine($"Key = {de.Key}, Value = {de.Value}");
  }
  WriteLine();

  // Вывести содержимое каждой таблицы, используя объект чтения данных
  foreach (DataTable dt in ds.Tables)
  {
    WriteLine($"=> {dt.TableName} Table:");
    // Вывести имена столбцов.
    for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
      Write($"{dt.Columns[curCol].ColumnName.Trim()}\t");
    }
    WriteLine("\n---------------------------------");
    // Вызвать новый вспомогательный метод.
    PrintTable(dt);
  }
}
```

В результате запуска приложения будет получен вывод, идентичный показанному ранее. Единственное отличие касается способа доступа к содержимому `DataTable`.

Сериализация объектов `DataTable` и `DataSet` в формате XML

Типы `DataSet` и `DataTable` поддерживают методы `WriteXml()` и `ReadXml()`. Метод `WriteXml()` позволяет сохранять содержимое объекта внутри локального файла (а также в любом типе, производном от `System.IO.Stream`) в виде документа XML.

Метод `ReadXml()` дает возможность восстановить состояние объекта `DataSet` (или `DataTable`) из указанного документа XML. Вдобавок типы `DataSet` и `DataTable` поддерживают методы `WriteXmlSchema()` и `ReadXmlSchema()`, предназначенные для сохранения или загрузки файла `*.xsd`.

В целях тестирования модифицируем код `Main()`, чтобы в нем вызывался следующий вспомогательный метод (которому передается единственный параметр типа `DataSet`):

```
static void SaveAndLoadAsXml(DataSet carsInventoryDS)
{
  // Сохранить этот объект DataSet в формате XML.
  carsInventoryDS.WriteXml("carsDataSet.xml");
  carsInventoryDS.WriteXmlSchema("carsDataSet.xsd");
  // Очистить объект DataSet.
  carsInventoryDS.Clear();
  // Загрузить объект DataSet из файла XML.
  carsInventoryDS.ReadXml("carsDataSet.xml");
}
```

Если открыть файл `carsDataSet.xml` (который находится в папке `bin\Debug` проекта), то можно обнаружить, что каждый столбец таблицы закодирован как элемент XML:

```
<?xml version="1.0" standalone="yes"?>
<Car_x0020_Inventory>
  <Inventory>
    <CarID>1</CarID>
    <Make>BMW</Make>
    <Color>Black</Color>
    <PetName>Hamlet</PetName>
  </Inventory>
  <Inventory>
    <CarID>2</CarID>
    <Make>Saab</Make>
    <Color>Red</Color>
    <PetName>Sea Breeze</PetName>
  </Inventory>
</Car_x0020_Inventory>
```

Двойной щелчок на сгенерированном файле `.xsd` (также находящемся в папке `bin\Debug`) внутри Visual Studio приводит к открытию встроенного редактора схемы XML (рис. A.3).

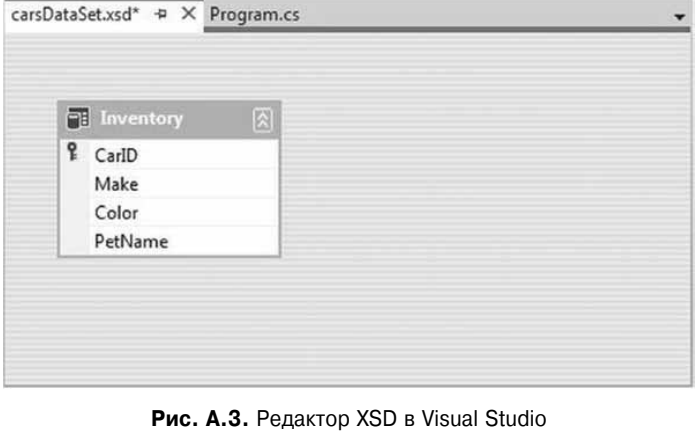

Рис. A.3. Редактор XSD в Visual Studio

На заметку! В приложении Б будет представлен API-интерфейс LINQ to XML, который является рекомендуемым способом манипулирования XML-данными в рамках платформы .NET.

Сериализация объектов `DataTable` и `DataSet` в двоичном формате

Содержимое объекта `DataSet` (или отдельного `DataTable`) можно также сохранять в компактном двоичном формате, что особенно полезно, когда объект `DataSet` должен передаваться за границы машины (в случае распределенного приложения). Один из недостатков представления данных в виде XML связан с тем, что его дескриптивная природа может привести к высоким накладным расходам при передаче.

Для сохранения объектов `DataTable` или `DataSet` в двоичном формате понадобится установить свойство `RemotingFormat` в `SerializationFormat.Binary`. После этого вполне ожидаемо можно использовать тип `BinaryFormatter` (см. главу 20). Рассмотрим следующий финальный метод проекта `SimpleDataSet` (важно не забыть об импортировании пространств имен `System.IO` и `System.Runtime.Serialization.Formatters.Binary`):

```
static void SaveAndLoadAsBinary(DataSet carsInventoryDS)
{
  // Установить флаг двоичной сериализации.
  carsInventoryDS.RemotingFormat = SerializationFormat.Binary;
  // Сохранить этот объект DataSet в двоичном виде.
  var fs = new FileStream("BinaryCars.bin", FileMode.Create);
  var bFormat = new BinaryFormatter();
  bFormat.Serialize(fs, carsInventoryDS);
  fs.Close();
  // Очистить объект DataSet.
  carsInventoryDS.Clear();
  // Загрузить объект DataSet из двоичного файла.
  fs = new FileStream("BinaryCars.bin", FileMode.Open);
  var data = (DataSet)bFormat.Deserialize(fs);
}
```

После вызова метода `SaveAndLoadAsBinary()` внутри `Main()` в папке `bin\Debug` можно будет найти файл `*.bin`. На рис. А.4 показано содержимое файла `BinaryCars.bin`.

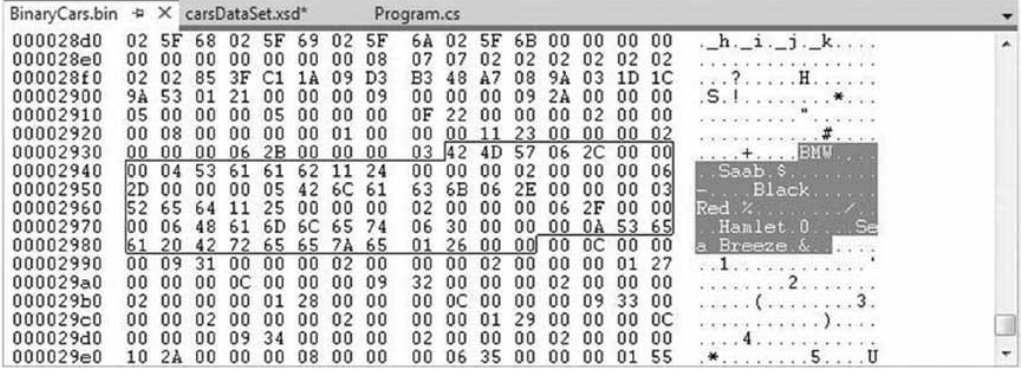

Рис. А.4. Объект `DataSet` сохранен в двоичном формате

Исходный код. Проект `SimpleDataSet` доступен в подкаталоге `Appendix_A`.

Привязка объектов DataTable к графическому пользовательскому интерфейсу Windows Forms

До сих пор вы видели, каким образом создавать, заполнять и проходить по содержимому объекта DataSet вручную с применением унаследованной объектной модели ADO.NET. Хотя знать, как это делается, довольно-таки важно, платформа .NET поставляется с многочисленными API-интерфейсами, которые обладают возможностью автоматической *привязки* данных к элементам пользовательского интерфейса.

Например, первоначальный набор инструментов для построения графических пользовательских интерфейсов .NET, Windows Forms, предлагает элемент управления по имени DataGridView, который обладает встроенной возможностью отображения содержимого объекта DataSet или DataTable с использованием всего нескольких строк кода. Инфраструктуры ASP.NET (API-интерфейс для разработки веб-приложений в .NET) и Windows Presentation Foundation также поддерживают понятие привязки данных. Благодаря этой книге вы научились привязывать данные к графическим элементам WPF и ASP.NET, однако в настоящем приложении будет применяться инфраструктура Windows Forms из-за присущей ей простой и понятной модели программирования.

На заметку! В следующем примере предполагается наличие у вас некоторого опыта использования Windows Forms для построения графических пользовательских интерфейсов. В противном случае можете просто открыть готовое решение (доступное в загружаемом коде примеров) и продолжить чтение.

Теперь нам предстоит построить приложение Windows Forms, которое будет отображать содержимое объекта DataTable внутри своего пользовательского интерфейса. Попутно вы увидите, каким образом фильтровать и изменять табличные данные. Вы также ознакомитесь с ролью объекта DataView.

Начнем с создания нового проекта приложения Windows Forms по имени WindowsFormsDataBinding. В окне Solution Explorer назначим начальному файлу вместо Form1.cs более подходящее имя MainForm.cs. В окне свойств изменим заголовок формы на Windows Forms Data Binding (Привязка данных в Windows Forms). Затем перетащим элемент DataGridView из панели инструментов Visual Studio на поверхность визуального конструктора (и в окне свойств переименуем его в carInventoryGridView с применением свойства (Name)). Можно заметить, что при добавлении элемента DataGridView на поверхность визуального конструктора в первый раз активизируется контекстное меню, которое позволяет подключиться к физическому источнику данных. Пока что проигнорируем этот аспект визуального конструктора, т.к. привязка объекта DataTable будет осуществляться программно. Наконец, поместим на поверхность визуального конструктора элемент Label с описательным текстом. Возможный внешний вид пользовательского интерфейса представлен на рис. A.5.

Заполнение DataTable из обобщенного List<T>

Подобно предыдущему примеру SimpleDataSet приложение WindowsFormsDataBinding будет конструировать объект DataTable, который содержит набор объектов DataColumn, представляющих разнообразные столбцы и строки данных. Но теперь строки будут заполняться с использованием переменной-члена обобщенного типа List<T>.

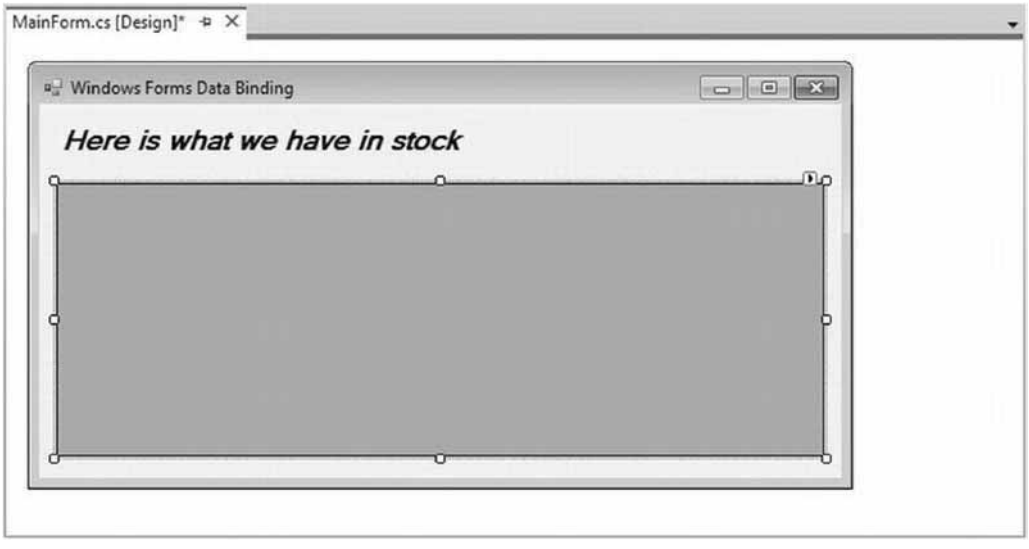

Рис. A.5. Начальный графический пользовательский интерфейс приложения Windows Forms

Для начала вставим в проект новый класс C# по имени `Car`, определенный следующим образом:

```
public class Car
{
  public int Id { get; set; }
  public string PetName { get; set; }
  public string Make { get; set; }
  public string Color { get; set; }
}
```

Внутри стандартного конструктора главной формы заполним переменную-член типа `List<T>` (`listCars`) набором новых объектов `Car`:

```
public partial class MainForm : Form
{
  // Коллекция объектов Car.
  List<Car> listCars = null;
  public MainForm()
  {
    InitializeComponent();

    // Заполнить список объектами Car.
    listCars = new List<Car>
    {
      new Car { Id = 1, PetName = "Chucky", Make = "BMW", Color = "Green" },
      new Car { Id = 2, PetName = "Tiny", Make = "Yugo", Color = "White" },},
      new Car { Id = 3, PetName = "Ami", Make = "Jeep", Color = "Tan" },
      new Car { Id = 4, PetName = "Pain Inducer", Make = "Caravan", Color = "Pink" },
      new Car { Id = 5, PetName = "Fred", Make = "BMW", Color = "Green" },
      new Car { Id = 6, PetName = "Sidd", Make = "BMW", Color = "Black" },
      new Car { Id = 7, PetName = "Mel", Make = "Firebird", Color = "Red" },
      new Car { Id = 8, PetName = "Sarah", Make = "Colt", Color = "Black" },
    };
  }
}
```

Добавим в класс `MainForm` новую переменную-член типа `DataTable` по имени `inventoryTable`:

```
public partial class MainForm : Form
{
  // Коллекция объектов Car.
  List<Car> listCars = null;

  // Складская информация.
  DataTable inventoryTable = new DataTable();
  ...
}
```

Добавим в класс `MainForm` новый вспомогательный метод `CreateDataTable()` и вызовем его в стандартном конструкторе класса `MainForm`:

```
void CreateDataTable()
{
  // Создать схему таблицы.
  var carIDColumn = new DataColumn("Id", typeof(int));
  var carMakeColumn = new DataColumn("Make", typeof(string));
  var carColorColumn = new DataColumn("Color", typeof(string));
  var carPetNameColumn = new DataColumn("PetName", typeof (string))
    { Caption = "Pet Name"};
  inventoryTable.Columns.AddRange(
    new[] { carIDColumn, carMakeColumn,carColorColumn, carPetNameColumn });

  // Пройти по элементам List<Car> для создания строк.
  foreach (var c in listCars)
  {
    var newRow = inventoryTable.NewRow();
    newRow["Id"] = c.Id;
    newRow["Make"] = c.Make;
    newRow["Color"] = c.Color;
    newRow["PetName"] = c.PetName;
    inventoryTable.Rows.Add(newRow);
  }

  // Привязать объект DataTable к carInventoryGridView.
  carInventoryGridView.DataSource = inventoryTable;
}
```

Реализация метода начинается с построения схемы `DataTable` путем создания четырех объектов `DataColumn` (ради простоты мы не заботимся об автоинкременте для поля `CarId` или об установке его как первичного ключа). Затем их можно добавить в коллекцию столбцов, доступную через переменную-член класса `DataTable`. Данные строк из коллекции `List<T>` сопоставляются с `DataTable` посредством итерационной конструкции `foreach` и собственной объектной модели ADO.NET.

Тем не менее, обратите внимание, что в последнем операторе внутри метода `CreateDataTable()` объект `inventoryTable` присваивается свойству `DataSource` объекта `DataGridView`. Установка этого единственного свойства — все, что требуется сделать для привязки `DataTable` к объекту `DataGridView` из Windows Forms. Указанный элемент управления графического пользовательского интерфейса внутренне читает коллекции строк и столбцов во многом подобно тому, как делалось в методе `PrintDataSet()` из примера `SimpleDataSet`. После запуска приложения должна появиться возможность просмотра содержимого объекта `DataTable` внутри элемента управления `DataGridView` (рис. A.6).

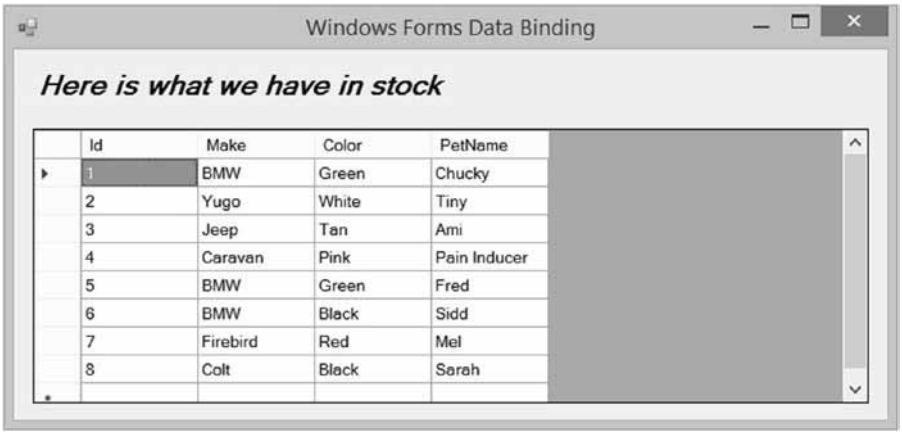

Рис. А.6. Привязка объекта `DataTable` к элементу `DataGridView` из Windows Forms

Удаление строк из `DataTable`

Теперь предположим, что необходимо обновить графический интерфейс, предоставив пользователю возможность удаления строки из находящегося в памяти объекта `DataTable`, который привязан к элементу управления `DataGridView`. Один из подходов предусматривает вызов метода `Delete()` объекта `DataRow`, который представляет удаляемую строку. В таком случае указывается индекс (или объект `DataRow`), соответствующий этой строке. Чтобы позволить пользователю выбрать строку для удаления, добавим на поверхность визуального конструктора элементы управления `TextBox` (по имени `txtRowToRemove`) и `Button` (по имени `btnRemoveRow`).

На рис. А.7 показан возможный внешний вид интерфейса после обновления (обратите внимание на помещение указанных двух элементов управления внутрь `GroupBox` с целью демонстрации взаимосвязи между ними).

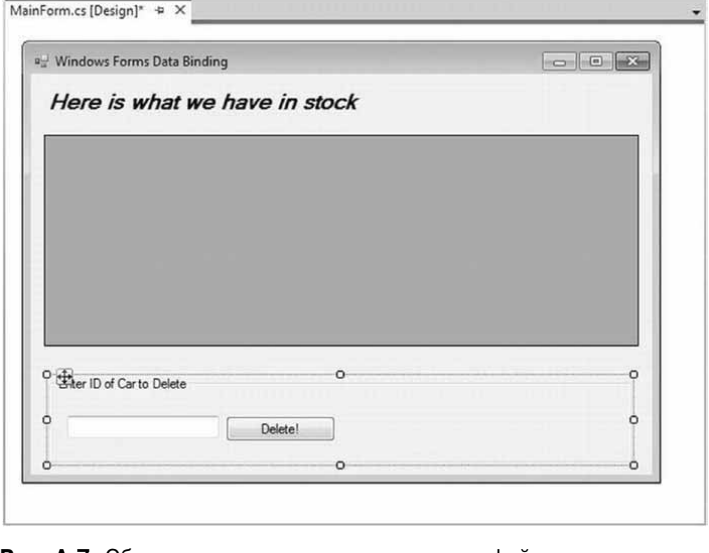

Рис. А.7. Обновление пользовательского интерфейса для включения возможности удаления строк из `DataTable`

Ниже приведен код обработчика события Click новой кнопки, в котором из находящегося в памяти объекта DataTable удаляется строка согласно введенного пользователем идентификатора автомобиля. Метод Select() класса DataTable позволяет указывать критерий поиска, моделируемый посредством обычного синтаксиса SQL. Возвращаемым значением является массив объектов DataRow, которые удовлетворяют критерию поиска.

```
// Удалить эту строку из DataRowCollection.
private void btnRemoveCar_Click (object sender, EventArgs e)
{
  try
  {
    // Найти корректную строку для удаления.
    DataRow[] rowToDelete =
        inventoryTable.Select($"Id={int.Parse(txtCarToRemove.Text)}");

    // Удалить ее.
    rowToDelete[0].Delete();
    inventoryTable.AcceptChanges();
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

Теперь можно запустить приложение и указать идентификатор удаляемого автомобиля. При удалении объектов DataRow из DataTable пользовательский интерфейс DataGridView обновляется немедленно, т.к. он привязан к состоянию объекта DataTable.

Выборка строк на основе критерия фильтрации

Многие приложения обработки данных требуют возможности просмотра небольшого подмножества данных из DataTable на основе указанного критерия фильтрации. Например, предположим, что необходимо отобразить только автомобили определенной марки (например, BMW) из объекта DataTable, хранящегося в памяти. Вы уже видели, как с помощью метода Select() класса DataTable находилась строка, подлежащая удалению, но его можно также применять для выборки подмножества записей в целях отображения.

Чтобы взглянуть на это в действии, снова изменим пользовательский интерфейс, предоставив пользователям возможность указывать строку, которая представляет интересующую модель автомобиля (рис. A.8), используя новые элементы управления TextBox (txtMakeToView) и Button (btnDisplayMakes).

Метод Select() имеет несколько перегруженных версий, которые предлагают разную семантику выборки. В простейшем случае передаваемый Select() параметр является строкой, содержащей условное выражение. Для начала рассмотрим логику обработчика события Click для новой кнопки:

```
private void btnDisplayMakes_Click(object sender, EventArgs e)
{
  // Построить фильтр на основе пользовательского ввода.
  string filterStr = $"Make='{txtMakeToView.Text}'";

  // Найти все строки, удовлетворяющие фильтру.
  DataRow[] makes = inventoryTable.Select(filterStr);
```

```
    // Показать полученные результаты.
    if (makes.Length == 0)
      MessageBox.Show("Sorry, no cars...", "Selection error!");
    else
    {
      string strMake = null;
      for (var i = 0; i < makes.Length; i++)
      {
        strMake += makes[i]["PetName"] + "\n";
      }
      // Вывести все результаты в окне сообщений.
      MessageBox.Show(strMake,$"We have {txtMakeToView.Text}s named:");
    }
  }
```

Здесь сначала создается простой фильтр на основе значения из связанного элемента управления `TextBox`. Если в текстовом поле фильтра указано `BMW`, то получится фильтр вида `Make = 'BMW'`. Передача такого фильтра методу `Select()` приводит к возвращению массива объектов `DataRow`, который представляет строки, удовлетворяющие данному фильтру (рис. А.9).

Логика фильтрации основана на стандартном синтаксисе SQL. Предположим что результаты предыдущего вызова `Select()` нужно получить в алфавитном порядке по столбцу `PetName`. К счастью, доступна перегруженная версия метода `Select()`, которая позволяет указывать критерий сортировки:

```
// Сортировать по PetName.
makes = inventoryTable.Select(filterStr, "PetName");
```

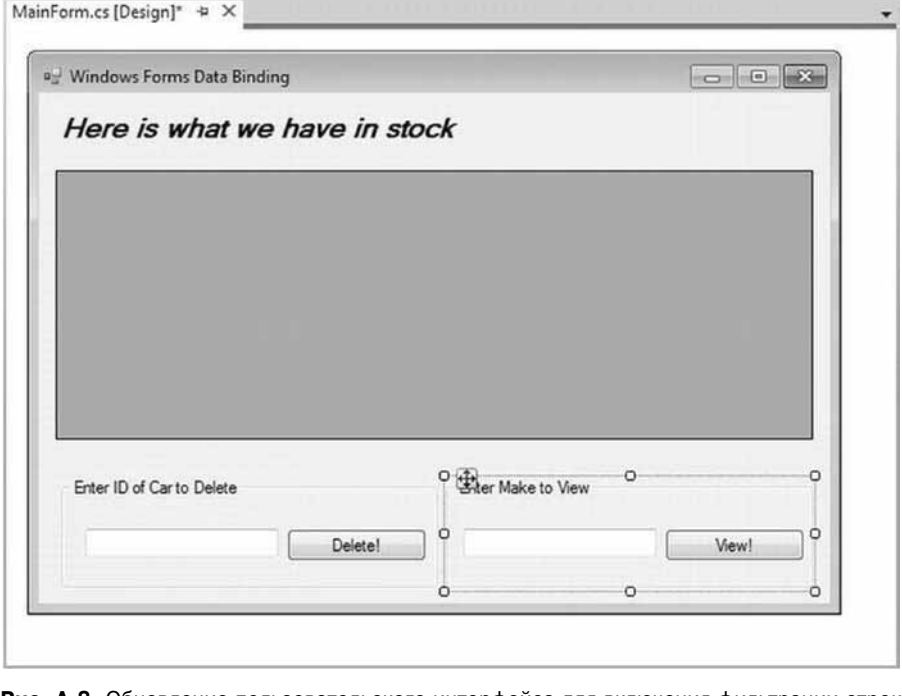

Рис. А.8. Обновление пользовательского интерфейса для включения фильтрации строк

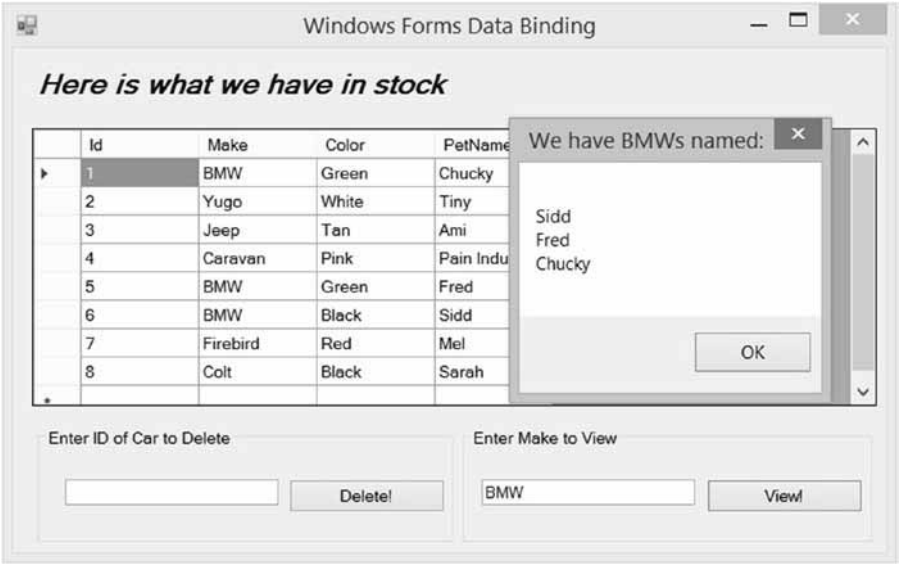

Рис. A.9. Отображение фильтрованных данных

Если необходимо вывести результаты в убывающем порядке, тогда понадобится вызвать метод `Select()` следующим образом:

```
// Возвратить результаты в убывающем порядке.
makes = inventoryTable.Select(filterStr, "PetName DESC");
```

В общем случае строка с критерием сортировки содержит имя столбца, за которым указана конструкция `ASC` (по возрастанию; принята по умолчанию) или `DESC` (по убыванию). Можно также указывать несколько столбцов, разделяя их запятыми. И, наконец, строка с критерием фильтрации может содержать любое количество операций отношения. Например, ниже показан вспомогательный метод, который выполняет поиск всех автомобилей со значением идентификатора больше 5:

```
private void ShowCarsWithIdGreaterThanFive()
{
  // Вывести дружественные имена всех автомобилей со значением ID больше 5.
  DataRow[] properIDs;
  string newFilterStr = "ID > 5";
  properIDs = inventoryTable.Select(newFilterStr);
  string strIDs = null;
  for(int i = 0; i < properIDs.Length; i++)
  {
    DataRow temp = properIDs[i];
    strIDs += $"{temp["PetName"]} is ID {temp["ID"]}\n";
  }
  MessageBox.Show(strIDs, "Pet names of cars where ID > 5");
}
```

Обновление строк в `DataTable`

Последний аспект `DataTable`, о котором вы должны знать — процесс обновления существующих строк новыми значениями. Один из подходов предусматривает сначала получение строки, удовлетворяющей заданному критерию фильтрации (или нескольких

таких строк), с помощью метода Select(). После получения интересующего объекта (или объектов) DataRow необходимо соответствующим образом модифицировать содержимое. Пусть на форме имеется новый элемент Button по имени btnChangeMakes, щелчок на котором приводит к поиску в DataTable всех строк, содержащих значение BMW в столбце Make. После получения таких строк значение Make изменяется на Yugo:

```
// Найти с помощью фильтра все строки, которые нужно отредактировать.
private void btnChangeMakes_Click(object sender, EventArgs e)
{
  // Удостовериться, что пользователь не изменил выбор.
  if (DialogResult.Yes !=
    MessageBox.Show("Are you sure?? BMWs are much nicer than Yugos!",
    "Please Confirm!", MessageBoxButtons.YesNo)) return;

  // Построить фильтр.
  string filterStr = "Make='BMW'";

  // Найти все строки, соответствующие фильтру.
  DataRow[] makes = inventoryTable.Select(filterStr);

  // Заменить BMW на Yugo.
  for (int i = 0; i < makes.Length; i++)
  {
    makes[i]["Make"] = "Yugo";
  }
}
```

Работа с типом DataView

Объект представления — это альтернативный вид таблицы (или набора таблиц). Например, с помощью Microsoft SQL Server можно создать представление для таблицы Inventory, которое возвращает новую таблицу, содержащую автомобили только указанного цвета. В ADO.NET тип DataView позволяет программным образом извлекать подмножество данных из DataTable в отдельный объект.

Серьезное преимущество наличия множества представлений одной и той же таблицы заключается в том, что их можно привязывать к разнообразным элементам управления графического пользовательского интерфейса (таким как DataGridView). Например, один DataGridView может быть привязан к объекту DataView, показывающему все автомобили из таблицы Inventory, тогда как другой можно сконфигурировать на отображение только автомобилей зеленого цвета.

Чтобы увидеть это в действии, добавим в текущий пользовательский интерфейс элемент управления DataGridView под названием dataGridColtsView и элемент Label с описанием. Затем определим переменную-член типа DataView по имени coltsOnlyView:

```
public partial class MainForm : Form
{
  // Представление DataTable.
  DataView yugosOnlyView;
  ...
}
```

Теперь создадим новый вспомогательный метод CreateDataView() и вызовем его в стандартном конструкторе главной формы сразу после того, как объект DataTable полностью сконструирован:

```
public MainForm()
{
  ...
```

```
  // Создать таблицу данных.
  CreateDataTable();

  // Создать представление.
  CreateDataView();
}
```

Ниже показана реализация нового вспомогательного метода. Обратите внимание, что конструктору `DataView` передается объект `DataTable`, который будет применяться для построения специального набора строк данных.

```
private void CreateDataView()
{
  // Установить таблицу, которая используется для создания этого представления.
  yugosOnlyView = new DataView(inventoryTable);

  // Сконфигурировать представление с применением фильтра.
  yugosOnlyView.RowFilter = "Make = 'Yugo'";

  // Привязать к новому элементу DataGridView.
  dataGridYugosView.DataSource = yugosOnlyView;
}
```

Как видите, класс `DataView` поддерживает свойство `RowFilter`, которое содержит строку с критерием фильтрации, используемым для извлечения интересующих строк. После создания представления установим соответствующим образом свойство `DataSource` элемента управления `DataGridView`. Завершенное приложение в действии показано на рис. A.10.

Рис. A.10. Отображение уникального представления данных

Исходный код. Проект `WindowsFormsDataBinding` доступен в подкаталоге `Appendix_A`.

Работа с адаптерами данных

Теперь, когда вы понимаете особенности ручного манипулирования объектами `DataSet` в ADO.NET, самое время переключить внимание на тему *объектов адаптеров данных*. Адаптер данных — это класс, применяемый для заполнения `DataSet` объектами `DataTable`; он также может отправлять модифицированные объекты `DataTable` обратно базе данных на обработку. В табл. A.8 кратко описаны основные члены базового класса `DbDataAdapter`, который является общим родительским классом для всех объектов адаптеров данных (скажем, `SqlDataAdapter` и `OdbcDataAdapter`).

Таблица A.8. Основные члены класса `DbDataAdapter`

Член	Описание
`Fill()`	Выполняет SQL-команду `SELECT` (указанную в свойстве `SelectCommand`) для выдачи запроса к базе данных и загрузки результирующих данных в объект `DataTable`

`SelectCommand`
`InsertCommand`
`UpdateCommand`
`DeleteCommand`
	Устанавливают SQL-команды, которые будут отправляться хранилищу данных, когда вызываются методы `Fill()` и `Update()`

`Update()`	Выполняет SQL-команды `INSERT`, `UPDATE` и `DELETE` (указанные в свойствах `InsertCommand`, `UpdateCommand` и `DeleteCommand`) для сохранения в базе данных изменений, произведенных в объекте `DataTable`

Обратите внимание, что в адаптере данных определены четыре свойства: `SelectCommand`, `InsertCommand`, `UpdateCommand` и `DeleteCommand`. При создании объекта адаптера данных для конкретного поставщика (например, `SqlDataAdapter`) можно передавать строку с текстом команды, которая будет применяться объектом команды `SelectCommand`.

Предполагая, что каждый из четырех объектов команд должным образом сконфигурирован, можно вызвать метод `Fill()` для получения объекта `DataSet` (или при желании одиночного объекта `DataTable`). Чтобы сделать это, адаптер данных должен выполнить SQL-оператор `SELECT`, указанный в свойстве `SelectCommand`.

Аналогично, если необходимо сохранить модифицированный объект `DataSet` (или `DataTable`) в базе данных, можно вызвать метод `Update()`, который будет использовать какой-то из оставшихся объектов команд в зависимости от состояния каждой строки в `DataTable` (вскоре мы обсудим все более подробно).

Одним из самых необычных аспектов работы с объектом адаптера данных является отсутствие потребности в открытии или закрытии подключения к базе данных. Взамен управление подключением к базе данных происходит автоматически. Однако адаптеру данных по-прежнему необходимо предоставить действительный объект подключения или строку подключения (на основе которой внутренне строится объект подключения), чтобы сообщить, с какой базой данных нужно взаимодействовать.

На заметку! Адаптер данных независим по своей природе. К нему можно присоединять на лету разные объекты подключения и объекты команд и извлекать информацию из широкого разнообразия баз данных. Например, единственный объект `DataSet` может содержать табличные данные, полученные от поставщиков данных SQL Server, Oracle и MySQL.

Простой пример адаптера данных

Следующий шаг заключается в добавлении новой функциональности в сборку библиотеки доступа к данным (AutoLotDAL.dll), которая была создана в главе 21. Начнем с простого примера, в котором объект DataSet заполняется одной таблицей с применением объекта адаптера данных ADO.NET.

Создадим новый проект консольного приложения по имени FillDataSetUsingSqlDataAdapter и импортируем в первоначальный файл кода C# пространства имен System.Data, System.Data.SqlClient и System.Collections. Далее модифицируем метод Main() следующим образом (в зависимости от того, как создавалась база данных AutoLot в главе 21, может понадобиться изменить строку подключения):

```
static void Main(string[] args)
{
  WriteLine("***** Fun with Data Adapters *****\n");

  // Жестко закодированная строка подключения.
  string connectionString =
    "Integrated Security = SSPI;Initial Catalog=AutoLot;" +
    @"Data Source=(local)\SQLEXPRESS2014";

  // Объект DataSet создается вызывающим кодом.
  DataSet ds = new DataSet("AutoLot");

  // Указать адаптеру текст команды Select и строку подключения.
  SqlDataAdapter adapter =
    new SqlDataAdapter("Select * From Inventory", connectionString);

  // Заполнить DataSet новой таблицей по имени Inventory.
  adapter.Fill(ds, "Inventory");

  // Отобразить содержимое DataSet.
  PrintDataSet(ds);
  ReadLine();
}
```

Обратите внимание, что адаптер данных конструируется с указанием строкового литерала, который будет отображен на SQL-оператор SELECT. Такое значение будет использоваться для внутреннего построения объекта команды, который позже можно получить через свойство SelectCommand.

Кроме того, создание экземпляра класса DataSet, который затем передается методу Fill(), является работой вызывающего кода. Дополнительно методу Fill() можно передать во втором аргументе строковое имя, которое будет применяться для установки свойства TableName нового объекта DataTable (если не указать имя таблицы, то адаптер данных назовет таблицу просто Table). В большинстве случаев имя, назначаемое DataTable, будет идентичным имени физической таблицы в реляционной базе данных; тем не менее, это не обязательно.

На заметку! Метод Fill() возвращает целое число, которое представляет количество строк, возвращенных запросом SQL.

Наконец, в методе Main() отсутствует явное открытие или закрытие подключения к базе данных. В методе Fill() любого адаптера данных заложена возможность открытия и закрытия подключения перед выходом из метода. Следовательно, когда объект DataSet передается методу PrintDataSet(), реализованному ранее в приложении и приведенному здесь в справочных целях, работа осуществляется с локальной копией автономных данных, не требуя обращений к СУБД для извлечения данных:

```
static void PrintDataSet(DataSet ds)
{
  // Вывести имя DataSet и любые расширенные свойства.
  WriteLine($"DataSet is named: {ds.DataSetName}");
  foreach (DictionaryEntry de in ds.ExtendedProperties)
  {
    WriteLine($"Key = {de.Key}, Value = {de.Value}");
  }
  WriteLine();
  foreach (DataTable dt in ds.Tables)
  {
    WriteLine($"=> {dt.TableName} Table:");
    // Вывести имена столбцов.
    for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
      Write(dt.Columns[curCol].ColumnName + "\t");
    }
    WriteLine("\n---------------------------------");
    // Вывести содержимое DataTable.
    for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
    {
      for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
      {
        Write(dt.Rows[curRow][curCol].ToString().Trim() + "\t");
      }
      WriteLine();
    }
  }
}
```

Отображение имен из базы данных на дружественные к пользователю имена

Как упоминалось ранее, администраторы баз данных обычно создают такие имена таблиц и столбцов, которые редко бывают дружественными в отношении конечных пользователей (скажем, `au_id`, `au_fname`, `au_lname` и т.д.). Хорошая новость в том, что объекты адаптеров данных поддерживают внутреннюю строго типизированную коллекцию (по имени `DataTableMappingCollection`) объектов `System.Data.Common.DataTableMapping`, к которой можно получать доступ через свойство `TableMappings` объекта адаптера данных.

При желании с помощью упомянутой коллекции объект `DataTable` можно информировать о том, какие *отображаемые имена* должны использоваться при выводе его содержимого. Предположим, что во время вывода вместо имени таблицы `Inventory` необходимо отображать `Current Inventory`, вместо имени столбца `CarId` — название `Car Id` (с пробелом), а вместо имени столбца `PetName` — строку `Name of Car`. Для этого перед вызовом метода `Fill()` объекта адаптера данных понадобится поместить показанный ниже код (а также импортировать пространство имен `System.Data.Common`, чтобы определение типа `DataTableMapping` стало доступным):

```
static void Main(string[] args)
{
  ...
  //Отобразить имена столбцов базы данных на дружественные к пользователю имена.
  DataTableMapping tableMapping =
    adapter.TableMappings.Add("Inventory", "Current Inventory");
```

```
  tableMapping.ColumnMappings.Add("CarId", "Car Id");
  tableMapping.ColumnMappings.Add("PetName", "Name of Car");
  dAdapt.Fill(ds, "Inventory");
  ...
}
```

Если снова запустить программу, то обнаружится, что метод `PrintDataSet()` теперь отображает дружественные имена объектов `DataTable` и `DataRow`, а не имена из схемы базы данных:

```
***** Fun with Data Adapters *****

DataSet is named: AutoLot

=> Current Inventory Table:
CarID Make   Color      Name of Car
---------------------------------
1     VW     Black      Zippy
2     Ford    Rust      Rusty
3     Saab    Black     Mel
4     Yugo    Yellow    Clunker
5     BMW    Black      Bimmer
6     BMW    Green      Hank
7     BMW    Pink       Pinkey
```

Исходный код. Проект `FillDataSetUsingSqlDataAdapter` доступен в подкаталоге `Appendix_A`.

Добавление функциональности автономного уровня в `AutoLotDAL.dll`

Чтобы продемонстрировать применение адаптера данных для передачи изменений в объекте `DataTable` базе данных, мы модифицируем созданную в главе 21 сборку `AutoLotDAL.dll`, включив в нее новое пространство имен (`AutoLotDisconnectedLayer`). Оно будет содержать новый класс `InventoryDALDC`, который использует адаптер данных для взаимодействия с объектом `DataTable`. Можно продолжить работу с проектом `AutoLotDAL`.

Определение начального класса

Добавим новую папку с помощью пункта меню Project⇨New Folder (Проект⇨Новая папка) и назначим ей имя `DisconnectedLayer`. Затем добавим в новую папку класс по имени `InventoryDALDC` (от **D**is**C**onnected), выбрав пункт меню Project⇨Add Class (Проект⇨Добавить класс). Далее снабдим тип класса в новом файле кода модификатором `public` и импортируем пространство имен `System.Data.SqlClient`.

В отличие от типа `InventoryDAL`, ориентированного на работу с подключением, новому классу не нужны специальные методы открытия/закрытия, поскольку адаптер данных обрабатывает все детали автоматически.

Начнем с добавления специального конструктора, который устанавливает закрытую переменную `string`, представляющую строку подключения. Кроме того, определим закрытую переменную-член `SqlDataAdapter`, которая будет конфигурироваться посредством вызова (пока еще не созданного) вспомогательного метода по имени `ConfigureAdapter()`, принимающего выходной параметр `SqlDataAdapter`:

```
namespace AutoLotDAL2.DisconnectedLayer
{
  public class InventoryDALDC
  {
    // Поля данных.
    private string _connectionString;
    private SqlDataAdapter _adapter = null;

    public InventoryDALDC(string connectionString)
    {
      _connectionString = connectionString;

      // Конфигурировать объект SqlDataAdapter.
      ConfigureAdapter(out _adapter);
    }

  }
 }
```

Конфигурирование адаптера данных с использованием SqlCommandBuilder

Перед применением адаптера данных для модификации таблиц в `DataSet` сначала необходимо присвоить свойствам `UpdateCommand`, `DeleteCommand` и `InsertCommand` допустимые объекты команд (до этого указанные свойства возвращают ссылки `null`).

Ручное конфигурирование объектов команд для свойств `InsertCommand`, `UpdateCommand` и `DeleteCommand` может повлечь за собой написание значительного объема кода, особенно если используются параметризированные запросы. Вспомните из главы 21, что параметризированные запросы позволяют строить операторы SQL с применением набора объектов параметров. Таким образом, если избран долгий путь, то метод `ConfigureAdapter()` можно было бы реализовать так, чтобы в нем создавались три новых объекта `SqlCommand`, каждый из которых содержал бы набор объектов `SqlParameter`. Затем готовые объекты можно было бы присвоить свойствам `UpdateCommand`, `DeleteCommand` и `InsertCommand` адаптера.

Среда Visual Studio предлагает несколько визуальных конструкторов, которые позаботятся о создании такого утомительного в написании кода. Визуальные конструкторы немного отличаются в зависимости от используемого API-интерфейса (например, Windows Forms, WPF или ASP.NET), но общая функциональность у них похожа. Примеры применения таких визуальных конструкторов неоднократно встречаются в книге, а позже в приложении используются некоторые визуальные конструкторы Windows Forms.

В настоящий момент не придется писать многочисленные операторы кода для полного конфигурирования адаптера данных; взамен работу можно значительно сократить, реализовав метод `ConfigureAdapter()` следующим образом:

```
private void ConfigureAdapter(out SqlDataAdapter adapter)
{
  // Создать адаптер и настроить SelectCommand.
  adapter = new SqlDataAdapter("Select * From Inventory", _connectionString);

  // Динамически получить остальные объекты команд
  // во время выполнения, используя SqlCommandBuilder.
  var builder = new SqlCommandBuilder(adapter);
}
```

Чтобы упростить конструирование объектов адаптеров данных, каждый поставщик данных ADO.NET от компании Microsoft предоставляет тип *построителя команд*. Тип `SqlCommandBuilder` автоматически генерирует значения для свойств `InsertCommand`,

UpdateCommand и DeleteCommand объекта SqlDataAdapter на основе начального объекта SelectCommand. Преимущество работы с ним в том, что отпадает необходимость вручную создавать все объекты SqlCommand и SqlParameter.

Здесь возникает очевидный вопрос: благодаря чему построитель команд способен создавать эти объекты команд на лету? Краткий ответ таков: благодаря метаданным. Когда во время выполнения вызывается метод Update() адаптера данных, связанный построитель команд читает информацию схемы базы данных и автоматически генерирует внутренние объекты команд вставки, удаления и обновления.

Понятно, что такие действия требуют дополнительных обращений к удаленной базе данных, т.е. при многократном применении типа SqlCommandBuilder в одном приложении его производительность ухудшится. Здесь мы минимизируем негативный эффект за счет вызова метода ConfigureAdapter() во время конструирования объекта InventoryDALDC и сохранения сконфигурированного объекта SqlDataAdapter для использования на протяжении всего времени существования InventoryDALDC.

В приведенном ранее коде объект построителя команд (SqlCommandBuilder) не использовался сверх того, что его конструктору был передан в качестве параметра объект адаптера данных. Как ни странно, это все, что должно быть сделано (в минимальном варианте). "За кулисами" тип SqlCommandBuilder конфигурирует адаптер данных с помощью остальных объектов команд.

Хотя вам может нравиться идея получить кое-что просто так, вы должны иметь в виду, что построители команд обладают рядом серьезных ограничений. В частности, построитель команд способен автоматически генерировать команды SQL для их применения адаптером данных, если удовлетворены все перечисленные ниже условия:

- SQL-команда SELECT взаимодействует только с одной таблицей (т.е. никаких соединений);
- единственная таблица имеет первичный ключ;
- таблица должна иметь столбец или столбцы, представляющие первичный ключ, который включается в SQL-оператор SELECT.

Учитывая способ построения базы данных AutoLot, такие ограничения не доставляют особых хлопот. Однако в производственной базе данных вам придется хорошо обдумать, будет ли этот тип хоть как-то полезен (если нет, то вспомните, что Visual Studio автоматически генерирует большой объем необходимого кода, используя разнообразные инструменты визуального конструирования баз данных, как вы увидите позже).

Реализация метода GetAllInventory()

Наш адаптер данных готов к применению. Первый метод нового класса будет просто вызывать метод Fill() объекта SqlDataAdapter для получения объекта DataTable, представляющего все записи в таблице Inventory базы данных AutoLot:

```
public DataTable GetAllInventory()
{
  DataTable inv = new DataTable("Inventory");
  _adapter.Fill(inv);
  return inv;
}
```

Реализация метода UpdateInventory()

Метод UpdateInventory() очень прост:

```
public void UpdateInventory(DataTable modifiedTable)
{
  _adapter.Update(modifiedTable);
}
```

Здесь объект адаптера данных проверяет значение `RowState` у каждой строки входного объекта `DataTable`. В зависимости от его значения (`RowState.Added`, `RowState.Deleted` или `RowState.Modified`) "за кулисами" задействуется подходящий объект команды.

Установка номера версии

Итак, создание логики второй версии библиотеки доступа к данным завершено. Хоть поступать так необязательно, установим номер версии библиотеки в 2.0.0.0 просто ради аккуратного ведения учета. Как объяснялось в главе 14, чтобы изменить версию сборки .NET, нужно дважды щелкнуть на значке Properties (Свойства) в окне Solution Explorer и затем щелкнуть на кнопке Assembly Information (Информация о сборке), расположенной на вкладке Application (Приложение). В открывшемся диалоговом окне укажем `2` для старшего номера версии сборки (за дополнительными сведениями обращайтесь в главу 14). Затем перекомпилируем приложение с целью обновления манифеста сборки.

Исходный код. Проект `AutoLotDAL2` доступен в подкаталоге `Appendix_A`.

Тестирование функциональности автономного уровня

В данный момент можно построить клиентское приложение для тестирования нового класса `InventoryDALDC`. Мы снова будем использовать API-интерфейс Windows Forms, чтобы отображать данные в графическом пользовательском интерфейсе. Создадим новый проект приложения Windows Forms по имени `InventoryDALDisconnectedGUI`. В окне Solution Explorer изменим первоначальное имя файла `Form1.cs` на `MainForm.cs` и установим свойство `Text` формы в `Simple GUI Front End to the Inventory Table` (Простое клиентское приложение с графическим пользовательским интерфейсом для таблицы `Inventory`). После создания проекта установим ссылку на обновленную сборку `AutoLotDAL.dll` (обязательно должна быть выбрана версия 2.0.0.0 сборки) и импортируем следующее пространство имен:

```
using AutoLotDAL2.DisconnectedLayer;
```

Главная форма приложения содержит элементы управления `Label` и `DataGridView` (по имени `inventoryGrid`), а также элемент управления `Button` (с именем `btnUpdateInventory`), который конфигурируется для обработки события `Click`. Вот определение формы:

```
public partial class MainForm : Form
{
  InventoryDALDC _dal = null;

  public MainForm()
  {
    InitializeComponent();

    string cnStr =
    @"Data Source=(local)\SQLEXPRESS2014;Initial Catalog=AutoLot;" +
     "Integrated Security=True;Pooling=False";

    // Создать объект доступа к данным.
    _dal = new InventoryDALDC(cnStr);
```

```
    // Заполнить элемент управления DataGridView.
    inventoryGrid.DataSource = _dal.GetAllInventory();
  }
  private void btnUpdateInventory_Click(object sender, EventArgs e)
  {
    // Получить модифицированные данные из DataGridView.
    DataTable changedDT = (DataTable)inventoryGrid.DataSource;
    try
    {
      // Зафиксировать изменения.
      _dal.UpdateInventory(changedDT);
      inventoryGrid.DataSource = _dal.GetAllInventory();
    }
    catch(Exception ex)
    {
      MessageBox.Show(ex.Message);
    }
  }
}
```

После создания объекта `InventoryDALDC` можно привязать объект `DataTable`, возвращенный в результате вызова метода `GetAllInventory()`, к элементу управления `DataGridView`. Когда пользователь щелкает на кнопке Update Database (Обновить базу данных), из `DataGridView` извлекается модифицированный объект `DataTable` (с помощью свойства `DataSource`) и передается методу `UpdateInventory()`.

Вот и все! Запустим приложение, добавим внутри сетки несколько новых строк и обновим/удалим ряд других строк. После щелчка на кнопке Update Database все изменения сохранятся в базе данных `AutoLot`. Из-за особенностей работы привязки данных в Windows Forms понадобится сбросить свойство `DataSource` сетки, чтобы изменения отобразились немедленно. При построении приложения с помощью Windows Presentation Foundation (WPF) вы заметите, что паттерн проектирования "Наблюдатель" (Observer) исправляет такое поведение.

Исходный код. Проект `InventoryDALDisconnectedGUI` доступен в подкаталоге `Appendix_A`.

Объекты `DataSet` с несколькими таблицами и отношения между данными

До сих пор все примеры, приведенные в приложении, имели дело с единственным объектом `DataTable`. Тем не менее, истинная мощь автономного уровня проявляется, когда объект `DataSet` содержит многочисленные взаимосвязанные объекты `DataTable`. В таком случае в коллекции `DataRelation` объекта `DataSet` можно определять любое количество объектов `DataRelation`, чтобы учесть все взаимозависимости между таблицами. На клиентском уровне объекты `DataRelation` можно применять для навигации между табличными данными, не обращаясь к сети.

На заметку! Вместо обновления сборки `AutoLotDAL.dll` для учета таблиц `Customers` и `Orders` в рассматриваемом примере вся логика доступа к данным выносится в новый проект Windows Forms. Однако смешивать логику пользовательского интерфейса с логикой обработки данных в производственном приложении не рекомендуется. В финальных примерах приложения будут задействованы разнообразные инструменты проектирования баз данных для отделения кода пользовательского интерфейса от кода взаимодействия с данными.

Начнем пример с создания нового проекта приложения Windows Forms по имени `MultitabledDataSetApp`. Графический пользовательский интерфейс приложения довольно прост (обратите внимание, что имя первоначального файла `Form1.cs` изменено на `MainForm.cs`, а свойство `Text` формы установлено в `AutoLot Database Manipulator` (Средство манипулирования базой данных `AutoLot`)). На рис. А.11 можно видеть три элемента управления `DataGridView` (`dataGridViewInventory`, `dataGridViewCustomers` и `dataGridViewOrders`), которые содержат данные, извлеченные из таблиц `Inventory`, `Orders` и `Customers` базы данных `AutoLot`. Кроме того, в интерфейсе предусмотрен элемент `Button` (`btnUpdateDatabase`), который позволяет отправить все внесенные в сетки изменения базе данных для обработки, используя объекты адаптеров данных.

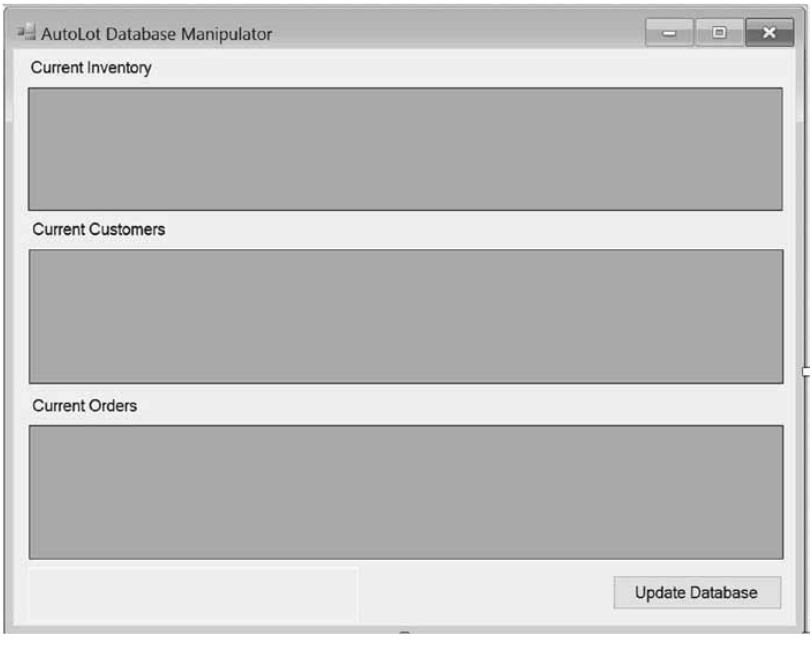

Рис. А.11. Первоначальный пользовательский интерфейс будет отображать содержимое всех таблиц базы данных `AutoLot`

Подготовка адаптеров данных

Для как можно большего упрощения кода доступа к данным в классе `MainForm` будут применяться объекты построителей команд, которые автоматически сгенерируют команды SQL для всех трех объектов `SqlDataAdapter` (по одному на каждую таблицу). Ниже показаны начальные изменения типа, производного от `Form` (важно не забыть об импортировании пространства имен `System.Data.SqlClient`):

```
public partial class MainForm : Form
{
  // Объект DataSet уровня формы.
  private DataSet _autoLotDs = new DataSet("AutoLot");

  // Использовать построители команд для упрощения конфигурирования
  // адаптеров данных.
  private SqlCommandBuilder _sqlCbInventory;
  private SqlCommandBuilder _sqlCbCustomers;
```

```
  private SqlCommandBuilder _sqlCbOrders;

  // Адаптеры данных (для каждой таблицы).
  private SqlDataAdapter _invTableAdapter;
  private SqlDataAdapter _custTableAdapter;
  private SqlDataAdapter _ordersTableAdapter;

  // Строка подключения уровня формы.
  private string _connectionString;
  ...
}
```

Конструктор делает всю черновую работу по созданию переменных-членов, относящихся к данным, и заполнению `DataSet`. В этом примере предполагается, что был создан файл `App.config`, который содержит корректную строку подключения (а также добавлена ссылка на сборку `System.Configuration.dll` и импортировано пространство имен `System.Configuration`):

```
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6" />
  </startup>
  <connectionStrings>
    <add name ="AutoLotSqlProvider" connectionString =
      "Data Source=(local)\SQLEXPRESS2014;
       Integrated Security=SSPI;Initial Catalog=AutoLot"
    />
  </connectionStrings>
</configuration>
```

Кроме того, обратите внимание на добавление вызова закрытого вспомогательного метода `BuildTableRelationship()`:

```
public MainForm()
{
  InitializeComponent();

  // Получить строку подключения.
  _connectionString =
    ConfigurationManager.ConnectionStrings["AutoLotSqlProvider"]
      .ConnectionString;

  // Создать объекты адаптеров.
  _invTableAdapter = new SqlDataAdapter(
     "Select * from Inventory", _connectionString);
  _custTableAdapter = new SqlDataAdapter(
     "Select * from Customers", _connectionString);
  _ordersTableAdapter = new SqlDataAdapter(
     "Select * from Orders", _connectionString);

  // Автоматически сгенерировать команды.
  _sqlCbInventory = new SqlCommandBuilder(_invTableAdapter);
  _sqlCbOrders = new SqlCommandBuilder(_ordersTableAdapter);
  _sqlCbCustomers = new SqlCommandBuilder(_custTableAdapter);

  // Заполнить таблицы в DataSet.
  _invTableAdapter.Fill(_autoLotDs, "Inventory");
  _custTableAdapter.Fill(_autoLotDs, "Customers");
  _ordersTableAdapter.Fill(_autoLotDs, "Orders");

  // Построить отношения между таблицами.
  BuildTableRelationship();
```

```
  // Привязаться к сеткам.
  dataGridViewInventory.DataSource = _autoLotDs.Tables["Inventory"];
  dataGridViewCustomers.DataSource = _autoLotDs.Tables["Customers"];
  dataGridViewOrders.DataSource = _autoLotDs.Tables["Orders"];
}
```

Построение отношений между таблицами

Вспомогательный метод `BuildTableRelationship()` выполняет всю рутинную работу по добавлению двух объектов `DataRelation` в объект `autoLotDS`. Вспомните из главы 21, что в базе данных `AutoLot` имеется несколько отношений "родительский–дочерний", которые можно учесть с помощью следующего кода:

```
private void BuildTableRelationship()
{
  // Создать объект отношения между данными CustomerOrder.
  DataRelation dr = new DataRelation("CustomerOrder",
     _autoLotDs.Tables["Customers"].Columns["CustID"],
     _autoLotDs.Tables["Orders"].Columns["CustID"]);
  _autoLotDs.Relations.Add(dr);

  // Создать объект отношения между данными InventoryOrder.
  dr = new DataRelation("InventoryOrder",
     _autoLotDs.Tables["Inventory"].Columns["CarID"],
     _autoLotDs.Tables["Orders"].Columns["CarID"]);
  _autoLotDs.Relations.Add(dr);
}
```

Обратите внимание, что при создании объекта `DataRelation` в первом параметре указывается дружественный строковый псевдоним (вы вскоре узнаете, почему это удобно). Кроме того, устанавливаются ключи, используемые для построения самого отношения. В коде видно, что родительская таблица (второй параметр конструктора) указывается перед дочерней таблицей (третий параметр конструктора).

Обновление таблиц базы данных

Теперь, когда объект `DataSet` заполнен информацией из источника данных, каждым объектом `DataTable` можно манипулировать локально. Запустим приложение и вставим, обновим или удалим значения в любом из трех элементов управления `DataGridView`. По готовности отправки изменений базе данных на обработку щелкнем на кнопке Update Database. К настоящему времени вам должно быть легко понять код в обработчике события `Click` указанной кнопки:

```
private void btnUpdateDatabase_Click(object sender, EventArgs e)
{
  try
  {
    _invTableAdapter.Update(_autoLotDs, "Inventory");
    _custTableAdapter.Update(_autoLotDs, "Customers");
    _ordersTableAdapter.Update(_autoLotDs, "Orders");
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

Запустим приложение и проведем различные обновления данных. При следующем запуске приложения должно обнаружиться, что содержимое сеток отражает все последние изменения.

Навигация между связанными таблицами

Теперь давайте посмотрим, как объект `DataRelation` позволяет программно перемещаться между связанными таблицами. Расширим наш пользовательский интерфейс путем добавления в него нового элемента управления `Button` (`btnGetOrderInfo`), связанного с ним `TextBox` (`txtCustID`) и `Label` с описательным текстом (для улучшения внешнего вида элементы управления можно сгруппировать внутри `GroupBox`).

На рис. A.12 показан возможный вид графического интерфейса приложения.

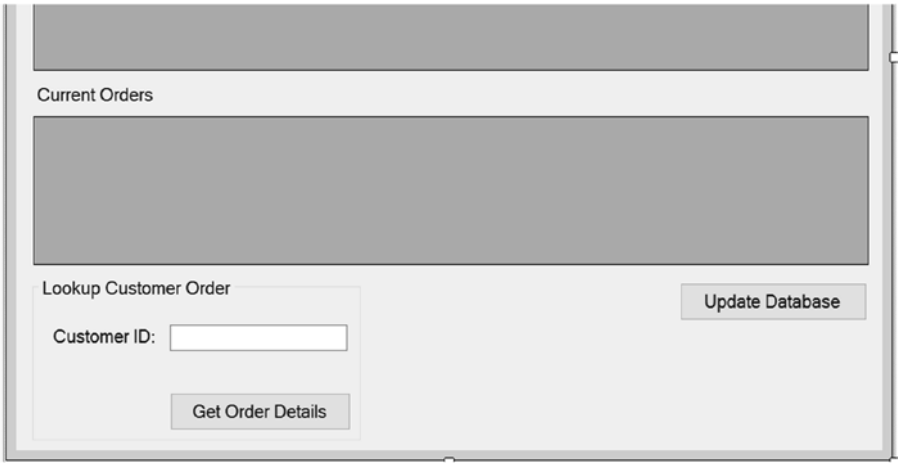

Рис. A.12. Обновленный пользовательский интерфейс предоставляет пользователю возможность поиска информации о заказах клиента

Обновленный пользовательский интерфейс позволяет пользователю ввести идентификатор клиента и извлечь связанную информацию о заказе этого клиента (имя, номер заказа и автомобиль). Информация форматируется в виде значения типа `string`, которое в конечном итоге отображается в окне сообщений. Взгляните на код обработчика события `Click` только что добавленной кнопки:

```
private void btnGetOrderInfo_Click(object sender, EventArgs e)
{
  string strOrderInfo = string.Empty;

  // Получить идентификатор клиента из текстового поля.
  int custID = int.Parse(txtCustID.Text);

  // На основе custID получить подходящую строку из таблицы Customers.
  var drsCust = _autoLotDs.Tables["Customers"].Select($"CustID = {custID}");
  strOrderInfo +=
    $"Customer {drsCust[0]["CustID"]}: {drsCust[0]["FirstName"].ToString().Trim()}
      {drsCust[0]["LastName"].ToString().Trim()}\n";

  // Перейти из таблицы Customers в таблицу Orders.
  var drsOrder = drsCust[0].GetChildRows(_autoLotDs.Relations["CustomerOrder"]);

  // Проход в цикле по всем заказам этого клиента.
  foreach (DataRow order in drsOrder)
```

```
  {
    strOrderInfo += $"----\nOrder Number: {order["OrderID"]}\n";
    // Получить автомобиль, на который ссылается этот заказ.
    DataRow[] drsInv =
      order.GetParentRows(_autoLotDs.Relations["InventoryOrder"]);
    // Получить информацию для (ОДНОГО) автомобиля из этого заказа.
    DataRow car = drsInv[0];
    strOrderInfo += $"Make: {car["Make"]}\n";
    strOrderInfo += $"Color: {car["Color"]}\n";
    strOrderInfo += $"Pet Name: {car["PetName"]}\n";
  }
  MessageBox.Show(strOrderInfo, "Order Details");
}
```

На рис. A.13 показан один из возможных результатов работы, когда задан идентификатор клиента 3 (ваш вывод может отличаться в зависимости от содержимого таблиц базы данных `AutoLot`).

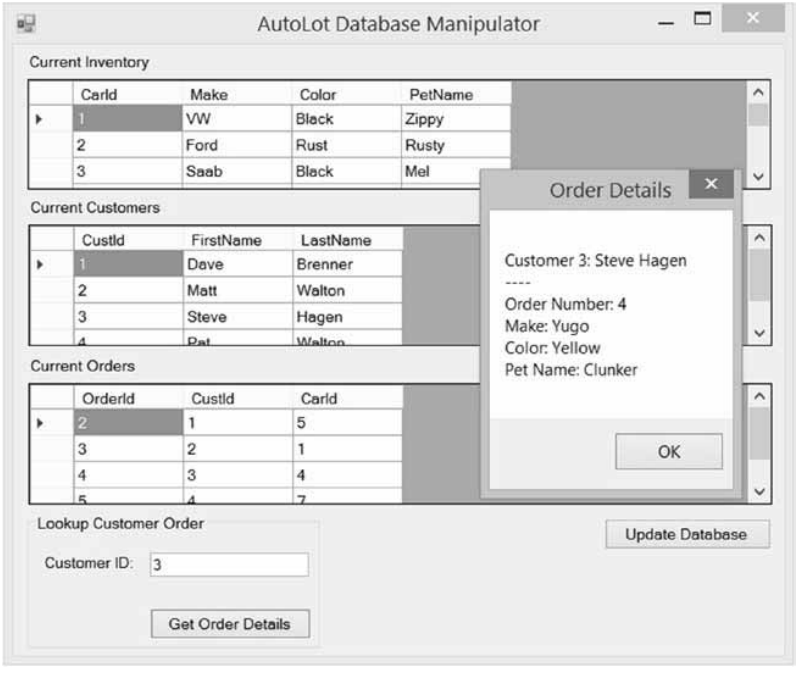

Рис. A.13. Навигация с помощью отношений между данными

Предыдущий пример должен был убедить вас в полезности класса `DataSet`. Учитывая, что объект `DataSet` полностью отключен от лежащего в основе источника данных, вы можете работать с находящейся в памяти копией данных и переходить от таблицы к таблице для выполнения необходимых обновлений, удалений или вставок, не обращаясь к базе данных. Завершив модификацию, вы можете отправить изменения хранилищу данных на обработку. Конечным результатом оказывается масштабируемое и надежное приложение.

Исходный код. Проект `MultitabledDataSetApp` доступен в подкаталоге `Appendix_A`.

Инструменты Windows Forms для визуального конструирования баз данных

Все приведенные до сих пор примеры предусматривали приличный объем непростой работы в том смысле, что всю логику доступа к данным приходилось писать вручную. В то время как значительная часть кода была вынесена в библиотеку .NET (`AutoLotDAL.dll`), перед взаимодействием с реляционной базой данных по-прежнему приходится вручную создавать разнообразные объекты имеющегося поставщика данных. Следующая задача заключается в исследовании инструментов визуального конструирования баз данных Windows Forms, которые могут создать за вас значительный объем кода доступа к данным.

Одним из способов применения интегрированных инструментов является использование визуальных конструкторов, поддерживаемых элементом управления `DataGridView` из Windows Forms. Проблема такого подхода в том, что инструменты визуального конструирования баз данных будут вставлять весь код доступа к данным прямо в кодовую базу графического пользовательского интерфейса! В идеале код, сгенерированный визуальным конструктором, лучше изолировать в выделенной библиотеке кода .NET, чтобы логику доступа к данным можно было многократно использовать во множестве проектов.

Тем не менее, полезно начать с выяснения того, как с помощью элемента `DataGridView` генерировать требуемый код доступа к данным, поскольку этот прием может быть полезен в небольших проектах и прототипах приложений. Затем вы узнаете, каким образом изолировать сгенерированный визуальным конструктором код в третьей версии библиотеки `AutoLotDAL.dll`.

Визуальное проектирование элемента управления `DataGridView`

С элементом управления `DataGridView` ассоциирован мастер, который может генерировать код доступа к данным. Начнем с создания нового проекта приложения Windows Forms по имени `DataGridViewDataDesigner`. В окне Solution Explorer переименуем первоначальную форму в `MainForm.cs`, установим ее свойство `Text` в `Windows Forms Data Wizards` (Мастера данных Windows Forms) и добавим к форме элемент управления `DataGridView` (по имени `inventoryDataGridView`). Когда элемент управления `DataGridView` выбран, справа от него должен открыться встроенный редактор (если он не открылся, нужно щелкнуть на кнопке с изображением треугольника в правом верхнем углу элемента управления). В раскрывающемся списке Choose Data Source (Выберите источник данных) щелкнем на ссылке Add Project Data Source (Добавить источник данных для проекта), как показано на рис. A.14.

Запустится мастер конфигурирования источников данных (Data Source Configuration Wizard). Он проведет через последовательность шагов, позволяющих выбрать и сконфигурировать источник данных, который затем будет привязан к `DataGridView`. На первом шаге мастер запрашивает тип источника данных, с которым необходимо взаимодействовать. Выберем вариант Database (База данных), как видно на рис. A.15, и щелкнем на кнопке Next (Далее).

На следующем шаге (который будет слегка отличаться в зависимости от выбора, произведенного на первом шаге) мастер запрашивает тип применяемой модели базы данных. Модель базы данных `Dataset` будет видна, только если в проект была добавлена инфраструктура Entity Framework. Выберем модель `Dataset` (рис. A.16).

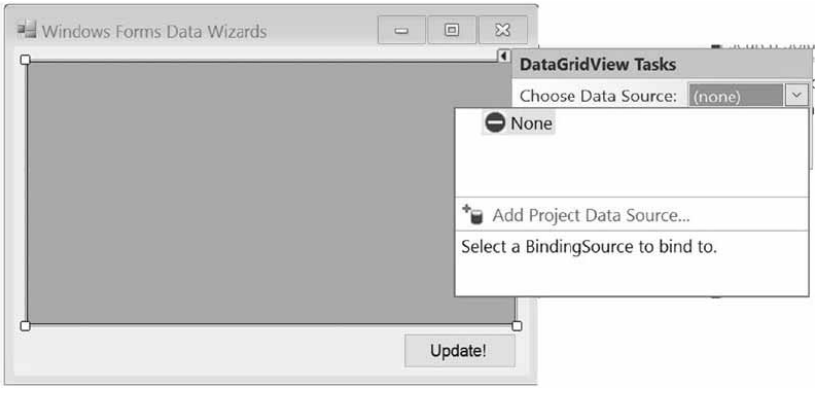

Рис. A.14. Редактор `DataGridView`

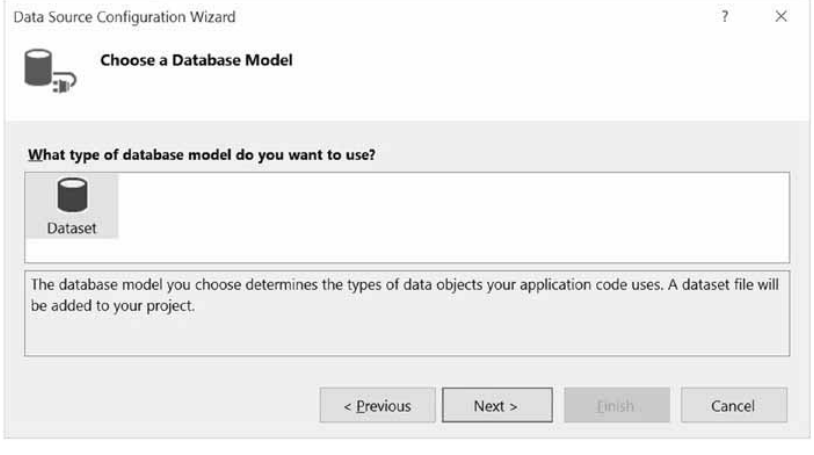

Рис. A.15. Выбор типа источника данных

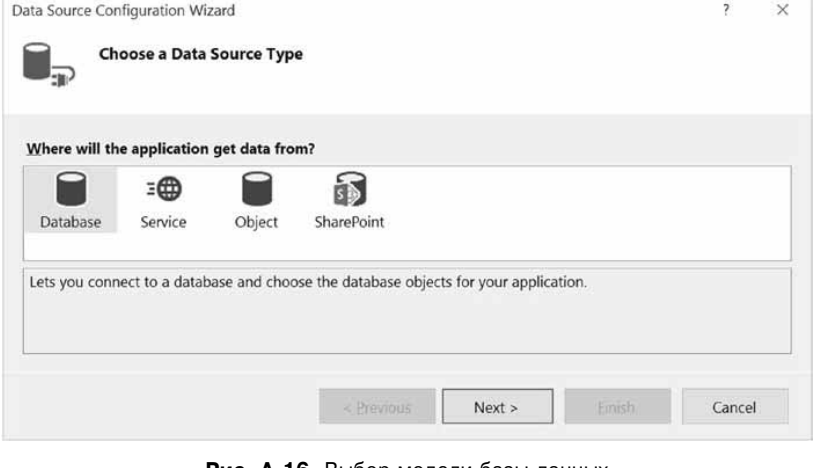

Рис. A.16. Выбор модели базы данных

Следующий шаг мастера позволяет сконфигурировать подключение к базе данных. Если база данных в текущий момент добавлена в окно Server Explorer, тогда она должна автоматически отображаться в раскрывающемся списке. В противном случае (или если необходимо подключиться к базе данных, ранее не добавленной в Server Explorer) понадобится щелкнуть на кнопке New Connection (Новое подключение). Результат выбора локального экземпляра базы данных `AutoLot` представлен на рис. A.17.

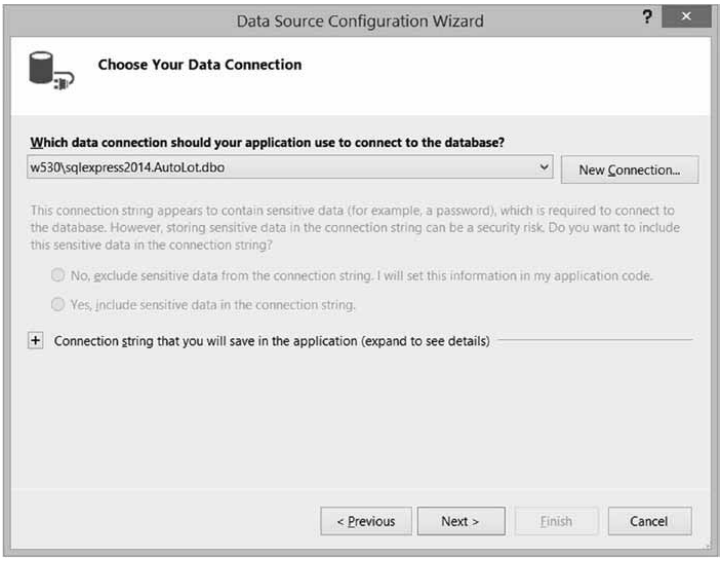

Рис. A.17. Выбор базы данных

На следующем шаге мастера выдается запрос о том, нужно ли сохранить строку подключения в конфигурационном файле приложения (рис. A.18). Отметим флажок, чтобы сохранить строку подключения, и щелкнем на кнопке Next.

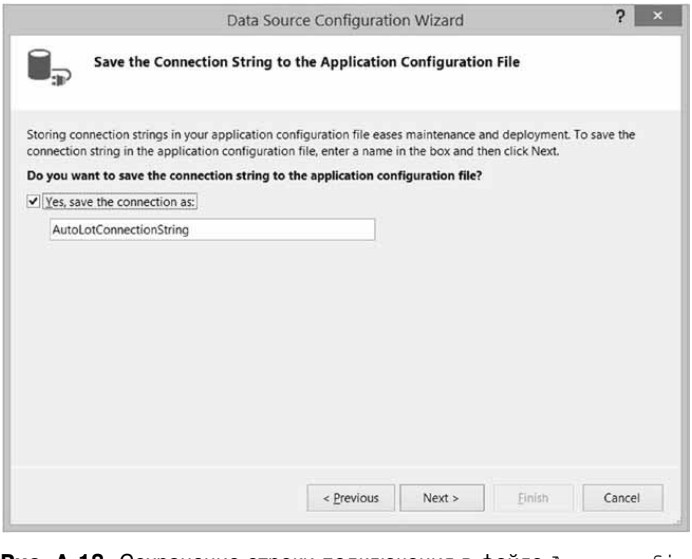

Рис. A.18. Сохранение строки подключения в файле `App.config`

На последнем шаге мастера выбираются объекты базы данных, которые будут учтены в автоматически сгенерированном классе `DataSet` и связанных адаптерах данных. Хотя можно было бы выбрать все объекты базы данных `AutoLot`, нас интересует только таблица `Inventory`. Изменим предлагаемое имя `DataSet` на `InventoryDataSet` (рис. А.19), отметим флажок возле таблицы `Inventory` и щелкнем на кнопке Finish (Готово).

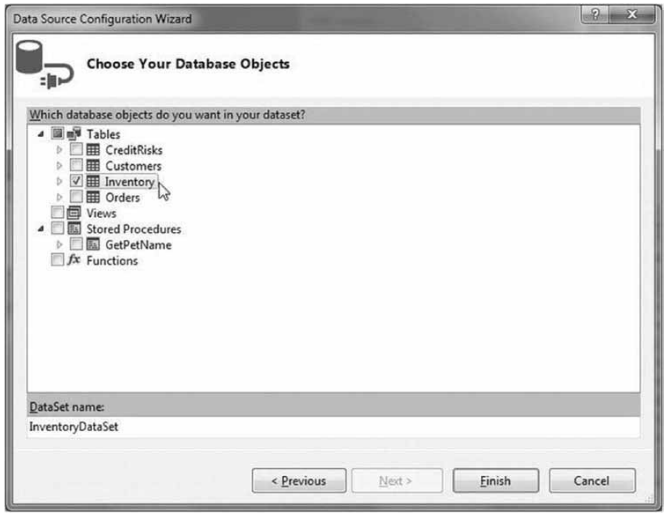

Рис. А.19. Выбор таблицы `Inventory`

После этого визуальный конструктор обновится во многих отношениях. Самое заметное изменение связано с тем, что элемент управления `DataGridView` теперь отображает схему таблицы `Inventory`, что видно по заголовкам столбцов. Кроме того, в нижней части визуального конструктора формы (в области, называемой *лотком с компонентами*) находятся три компонента: `DataSet`, `BindingSource` и `TableAdapter` (рис. А.20).

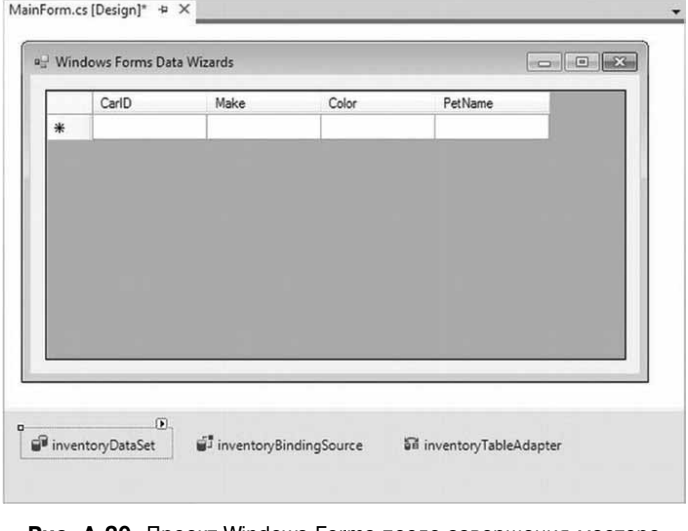

Рис. А.20. Проект Windows Forms после завершения мастера конфигурирования источников данных

Теперь приложение можно запустить и сетка заполнится записями из таблицы Inventory. Конечно же, никакой магии здесь нет. Просто IDE-среда автоматически создала порядочный объем кода и настроила элемент управления типа сетки для последующего использования. Давайте проанализируем такой автоматически сгенерированный код.

Сгенерированный файл App.config

В окне Solution Explorer можно заметить, что проект содержит файл App.config, в котором имеется элемент <connectionStrings> с несколько необычным именем:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
  </configSections>
  <connectionStrings>
    <add name=
    "DataGridViewDataDesigner.Properties.Settings.AutoLotConnectionString"
   connectionString=
"Data Source=.\SQLEXPRESS2014;Initial Catalog=AutoLot;Integrated Security=True"
       providerName="System.Data.SqlClient" />
  </connectionStrings>
    <startup>
      <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6" />
    </startup>
</configuration>
```

Автоматически сгенерированный объект адаптера данных (о котором будет рассказано ниже) применяет длинное значение:

```
"DataGridViewDataDesigner.Properties.Settings.AutoLotConnectionString"
```

Исследование строго типизированного класса DataSet

В дополнение к конфигурационному файлу мастер генерирует так называемый *строго типизированный класс* ***DataSet***. Этим термином обозначается специальный класс, расширяющий DataSet и открывающий доступ к нескольким членам, которые позволяют взаимодействовать с базой данных, используя интуитивно понятную объектную модель. Например, строго типизированные объекты DataSet содержат свойства, которые отображаются непосредственно на имена таблиц базы данных. Таким образом, свойство Inventory можно применять для обращения к строкам и столбцам базы напрямую, не углубляясь в коллекцию таблиц с использованием свойства Tables.

Если вставить в проект новый файл диаграммы классов, то можно заметить, что мастер создал класс по имени InventoryDataSet. В нем определен набор членов, из которых наиболее важным является свойство Inventory (рис. A.21).

Двойной щелчок на файле InventoryDataSet.xsd в окне Solution Explorer приводит к загрузке визуального конструктора наборов данных Visual Studio (который более подробно рассматривается далее в приложении). Если щелкнуть правой кнопкой мыши на поверхности этого конструктора и выбрать в контекстном меню пункт View Code (Просмотреть код), тогда отобразится следующее практически пустое определение частичного класса:

```
partial class InventoryDataSet
{
}
```

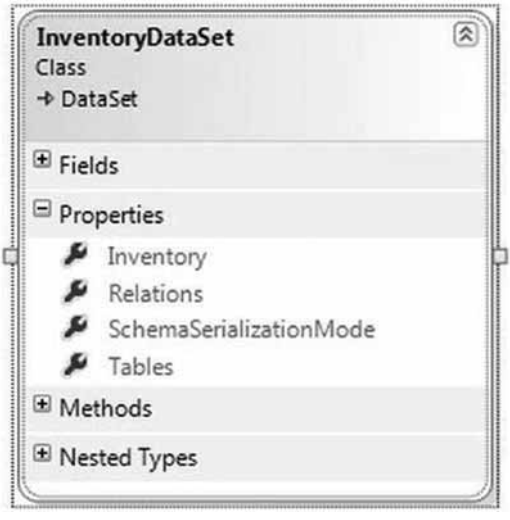

Рис. A.21. Мастер конфигурирования источников данных создал строго типизированный класс `DataSet`

При необходимости к данному определению частичного класса допускается добавлять специальные члены, однако реальное действие происходит в обслуживаемом конструктором файле `InventoryDataSet.Designer.cs`, открыв который в Solution Explorer, можно заметить, что `InventoryDataSet` расширяет родительский класс `DataSet`. Взгляните на показанный ниже фрагмент кода с комментариями добавленными для ясности:

```
// Весь этот код сгенерирован конструктором!
public partial class InventoryDataSet : global::System.Data.DataSet
{
  // Переменная-член типа InventoryDataTable.
  private InventoryDataTable tableInventory;

  // Каждый конструктор вызывает вспомогательный метод по имени InitClass().
  public InventoryDataSet()
  {
    ...
    this.InitClass();
    ...
  }

  // Метод InitClass() подготавливает DataSet и добавляет
  // InventoryDataTable в коллекцию Tables.
  private void InitClass()
  {
    this.DataSetName = "InventoryDataSet";
    this.Prefix = "";
    this.Namespace = "http://tempuri.org/InventoryDataSet.xsd";
    this.EnforceConstraints = true;
    this.SchemaSerializationMode =
      global::System.Data.SchemaSerializationMode.IncludeSchema;
    this.tableInventory = new InventoryDataTable();
    base.Tables.Add(this.tableInventory);
  }
```

```
  // Свойство Inventory, предназначенное только для чтения,
  // возвращает переменную-член InventoryDataTable.
  public InventoryDataTable Inventory
  {
    get { return this.tableInventory; }
  }
}
```

Обратите внимание, что в строго типизированном классе `DataSet` есть переменная-член, которая относится к *строго типизированному классу* `DataTable` — в данном случае классу `InventoryDataTable`. Конструктор строго типизированного класса `DataSet` вызывает закрытый метод инициализации `InitClass()`, который добавляет экземпляр строго типизированного `DataTable` в коллекцию `Tables` объекта `DataSet`. Кроме того, реализация свойства `Inventory` возвращает переменную-член `InventoryDataTable`.

Исследование строго типизированного класса `DataTable`

Возвратимся к файлу диаграммы классов и раскроем узел Nested Types (Вложенные типы) для класса `InventoryDataSet`. Здесь будет находиться строго типизированный класс `DataTable` по имени `InventoryDataTable` и *строго типизированный* класс `DataRow` по имени `InventoryRow`.

В классе `InventoryDataTable` (который имеет тот же самый тип, что и рассмотренная переменная-член строго типизированного `DataSet`) определен набор свойств, основанных на именах столбцов физической таблицы `Inventory` (`CarIdColumn`, `ColorColumn`, `MakeColumn` и `PetNameColumn`), а также специальный индексатор и свойство `Count` для получения текущего количества записей.

Более интересно то, что в этом строго типизированном классе `DataTable` определены методы, которые позволяют вставлять, находить и удалять строки в таблице с применением строго типизированных членов (удобная альтернатива ручной навигации по индексаторам `Rows` и `Columns`). Например, метод `AddInventoryRow()` предназначен для добавления новой строки к находящейся в памяти таблице, `FindByCarId()` — для поиска в таблице по первичному ключу, а `RemoveInventoryRow()` — для удаления строки из строго типизированной таблицы (рис. A.22).

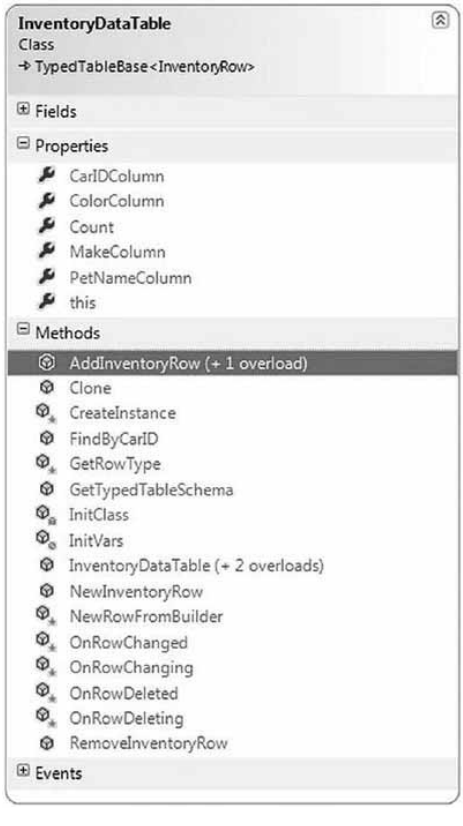

Рис. A.22. Строго типизированный класс `DataTable`, вложенный в строго типизированный класс `DataSet`

Исследование строго типизированного класса `DataRow`

Строго типизированный класс `DataRow`, также вложенный внутрь строго типизированного класса `DataSet`, расширяет класс `DataRow` и открывает доступ к свойствам, которые отображаются прямо на схему

таблицы `Inventory`. Кроме того, визуальный конструктор баз данных создал метод `IsPetNameNull()`, который проверяет, содержит ли данный столбец значение (рис. А.23).

Исследование строго типизированного адаптера данных

Строгая типизация для автономных типов является серьезным преимуществом, которое дает использование мастера конфигурирования источников данных, т.к. создание этих классов вручную может оказаться утомительным (хотя и вполне посильным) занятием. Тот же самый мастер способен даже генерировать объект специального адаптера данных, который может заполнять и обновлять объекты `InventoryDataSet` и `InventoryDataTable` в строго типизированной манере. Отыщем в окне визуального конструктора классов класс `InventoryTableAdapter` и просмотрим сгенерированные для него члены (рис. А.24).

Автоматически сгенерированный тип `InventoryTableAdapter` поддерживает коллекцию объектов `SqlCommand` (с доступом к ним через свойство `CommandCollection`), каждый из которых имеет полностью заполненный набор объектов `SqlParameter`. Вдобавок этот специальный адаптер данных предлагает набор свойств для извлечения лежащих в основе объектов подключения, транзакции и адаптера данных, а также свойство для получения массива, представляющего все типы команд.

Завершение приложения Windows Forms

Если внимательно изучить обработчик события `Load` в производном от формы типе (другими словами, отобразить код `MainForm.cs` и найти метод `MainForm_Load()`), то обнаружится, что в самом начале вызывается метод `Fill()` специального адаптера данных таблицы с передачей ему специального объекта `DataTable`, поддерживаемого специальным `DataSet`:

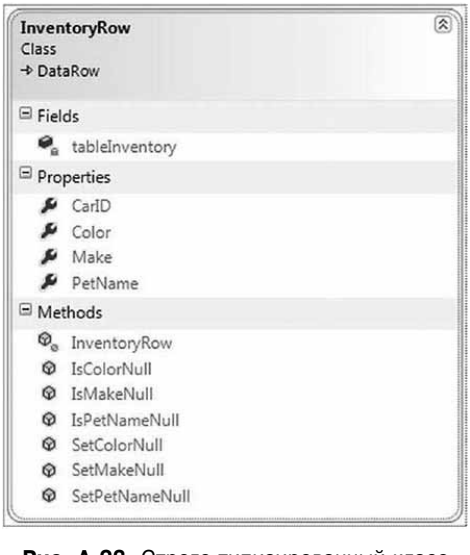

Рис. А.23. Строго типизированный класс `DataRow`

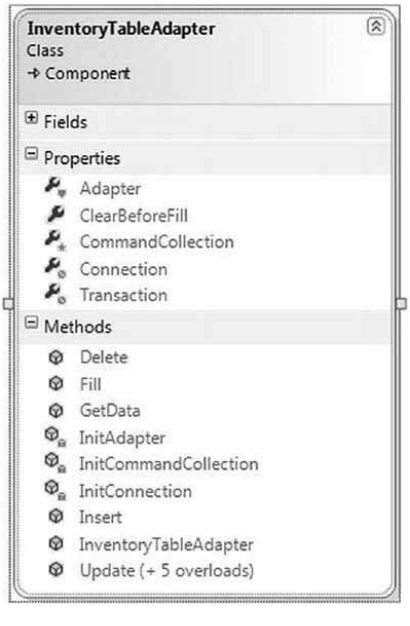

Рис. А.24. Настроенный адаптер данных, который оперирует со строго типизированными классами `DataSet` и `DataTable`

```
private void MainForm_Load(object sender, EventArgs e)
{
  this.inventoryTableAdapter.Fill(this.inventoryDataSet.Inventory);
}
```

Тот же самый специальный объект адаптера данных можно применять для обновления сетки изменениями, произошедшими в данных. Добавим к пользовательскому интерфейсу формы элемент управления `Button` (по имени `btnUpdateInventory`). Затем создадим для него обработчик события `Click` со следующим кодом:

```
private void btnUpdateInventory_Click(object sender, EventArgs e)
{
  try
  {
    // Сохранить в базе данных изменения, внесенные в таблицу Inventory.
    this.inventoryTableAdapter.Update(this.inventoryDataSet.Inventory);
  }
  catch(Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
  // Получить актуальную копию данных для сетки.
  this.inventoryTableAdapter.Fill(this.inventoryDataSet.Inventory);
}
```

Снова запустим приложение, добавим, удалим или обновим записи, отображаемые в сетке, а затем щелкнем на кнопке Update Database (Обновить базу данных). Теперь видно, что все изменения учтены.

Итак, рассмотренный пример продемонстрировал, насколько полезным может быть визуальный конструктор элемента управления `DataGridView`. Он позволяет работать со строго типизированными данными и генерирует большую часть логики, необходимой для взаимодействия с базой данных. Очевидная проблема заключается в том, что результирующий код тесно связан с окном, в котором он используется. В идеальном случае такая разновидность кода должна находиться в сборке `AutoLotDAL.dll` (или в какой-то другой библиотеке доступа к данным). Тем не менее, вас может интересовать, каким образом задействовать код, который генерируется мастером, ассоциированным с элементом управления `DataGridView`, в проекте библиотеки классов, поскольку по умолчанию визуальный конструктор форм в нем отсутствует.

Исходный код. Проект `DataGridViewDataDesigner` доступен в подкаталоге `Appendix_A`.

Изоляция строго типизированного кода работы с базой данных в библиотеке классов

К счастью, активизировать инструменты визуального проектирования данных среды Visual Studio можно в любой разновидности проекта (с пользовательским интерфейсом или без), не копируя крупные фрагменты кода между проектами. Чтобы увидеть их в действии, мы добавим в `AutoLotDAL.dll` дополнительную функциональность. Можно продолжить работу с существующим проектом. В загружаемом коде примеров доступен отдельный проект под названием `AutoLotDAL3`.

Создадим внутри папки проекта новую папку `DataSets` и поместим в нее строго типизированный класс `DataSet` (по имени `AutoLotDataSet.xsd`) с применением пункта

меню Project⇨Add New Item (Проект⇨Добавить новый элемент). Чтобы быстро найти тип элемента `DataSet`, в диалоговом окне New Item (Новый элемент) понадобится выбрать раздел Data (Данные), как показано на рис. А.25.

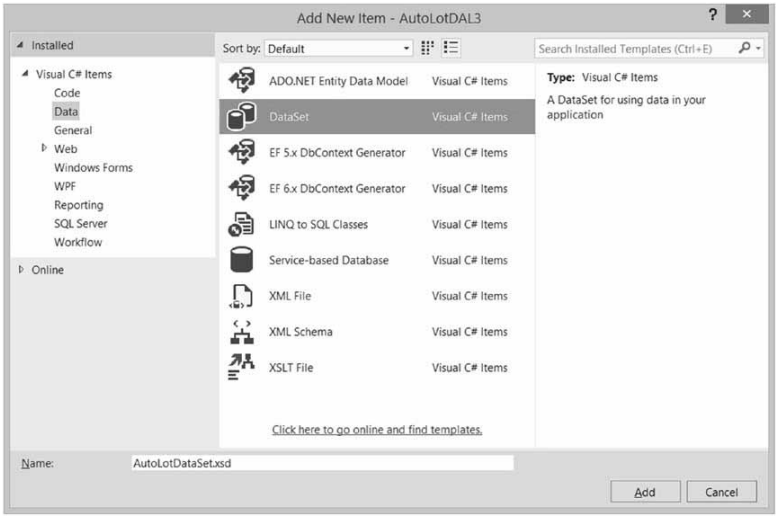

Рис. А.25. Вставка нового строго типизированного класса `DataSet`

Откроется пустая поверхность визуального конструктора наборов данных. Теперь с помощью окна Server Explorer можно подключиться к нужной базе данных (подключение к `AutoLot` уже должно существовать) и перетащить на поверхность все таблицы и хранимые процедуры, которые требуются в `DataSet`. На рис. А.26 видно, что учтены все специфические аспекты базы `AutoLot`, а отношения между таблицами реализованы автоматически (в текущем примере таблица `CreditRisk` не перетаскивалась).

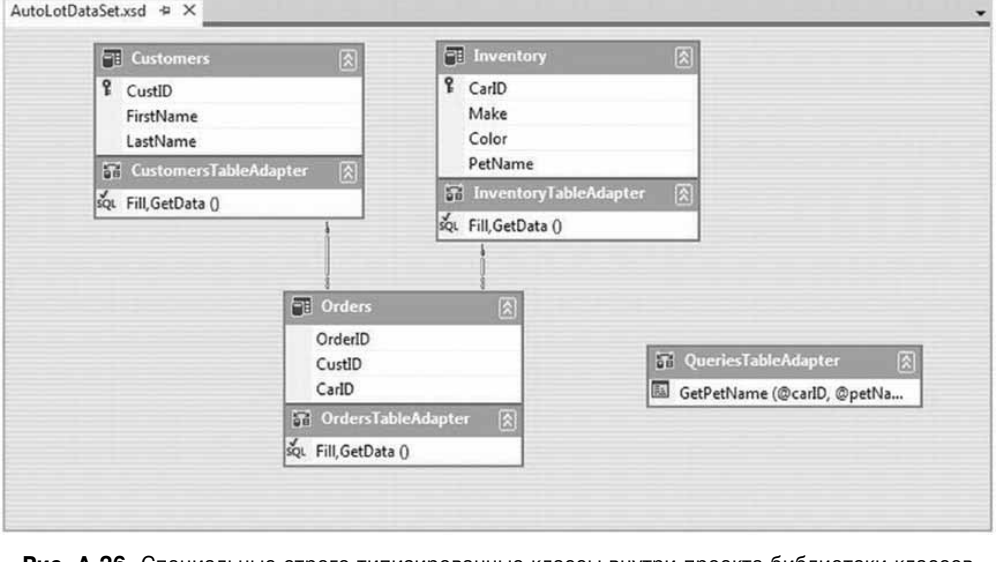

Рис. А.26. Специальные строго типизированные классы внутри проекта библиотеки классов

Просмотр сгенерированного кода

Визуальный конструктор наборов данных создал ту же самую разновидность кода, что и мастер `DataGridView` в предыдущем примере приложения Windows Forms. Однако на этот раз были задействованы таблицы `Inventory`, `Customers` и `Orders`, а также хранимая процедура `GetPetName`, и потому сгенерированных классов получилось намного больше. По существу каждая таблица базы данных, помещенная на поверхность визуального конструктора, дала в результате классы `DataTable`, `DataRow` и адаптера данных, которые содержатся в строго типизированном `DataSet`.

Строго типизированные классы `DataSet`, `DataTable` и `DataRow` будут помещены в корневое пространство имен проекта (`AutoLotDAL`). Специальные адаптеры таблиц будут находиться во вложенном пространстве имен. Просмотреть все сгенерированные типы проще всего с использованием окна Class View (Представление классов), которое открывается через меню View (Вид) в Visual Studio (рис. A.27).

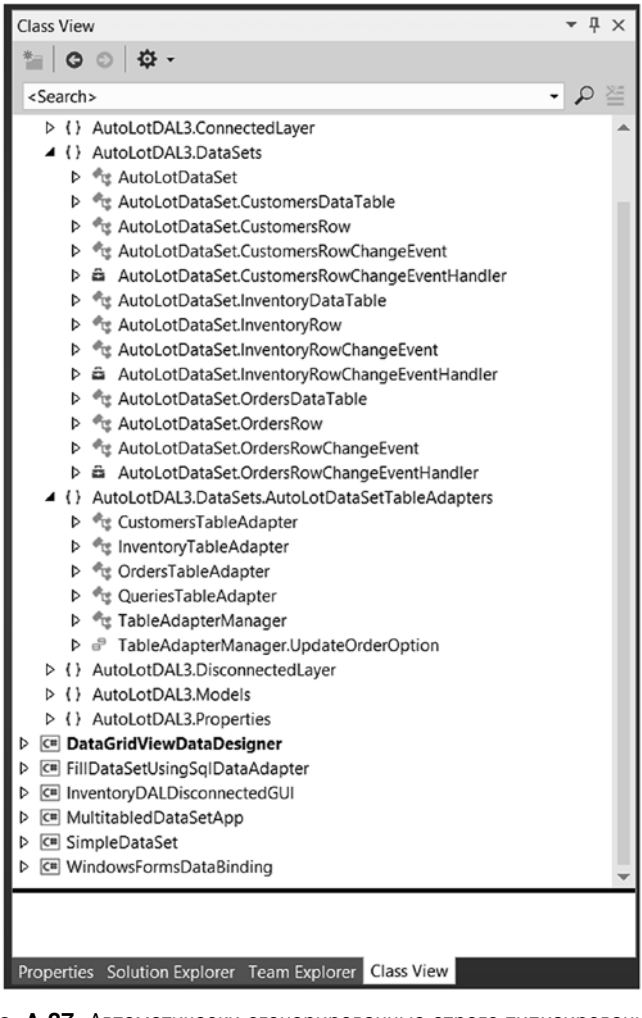

Рис. A.27. Автоматически сгенерированные строго типизированные классы для базы данных `AutoLot`

Ради завершенности можно открыть окно Properties (Свойства) в Visual Studio (см. главу 14) и изменить версию последней модификации сборки AutoLotDAL.dll на 3.0.0.0.

Исходный код. Проект AutoLotDAL3 доступен в подкаталоге Appendix_A.

Выборка данных с помощью сгенерированного кода

Полученные к настоящему моменту строго типизированные классы можно применять в любом приложении .NET, которому необходимо взаимодействовать с базой данных AutoLot. Чтобы удостовериться в понимании всех основных механизмов, создадим консольное приложение по имени StronglyTypedDataSetConsoleClient. Добавим в него ссылку на последнюю версию сборки AutoLotDAL3.dll, импортируем в первоначальный файл кода C# пространства имен AutoLotDAL3.DataSets и AutoLotDAL3.DataSets.AutoLotDataSetTableAdapters, а также добавим оператор using static System.Console;.

Ниже приведен код метода Main(), в котором объект InventoryTableAdapter используется для выборки всех данных из таблицы Inventory. Обратите внимание, что здесь нет необходимости указывать строку подключения, т.к. информация о ней теперь является частью строго типизированной объектной модели. После заполнения таблицы результаты выводятся с применением вспомогательного метода по имени PrintInventory(). Манипулировать строго типизированным DataTable можно точно так же, как делалось с "обычным" объектом DataTable, используя коллекции Rows и Columns.

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Strongly Typed DataSets *****\n");

    // Вызываемый код создает объект DataSet.
    var table = new AutoLotDataSet.InventoryDataTable();

    // Информировать адаптер о команде Select и подключении.
    var adapter = new InventoryTableAdapter();

    // Заполнить объект DataSet новой таблицей по имени Inventory.
    adapter.Fill(table);

    PrintInventory(table);
    Console.ReadLine();
  }

  static void PrintInventory(AutoLotDataSet.InventoryDataTable dt)
  {
    // Вывести имена столбцов.
    for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
      Write(dt.Columns[curCol].ColumnName + "\t");
    }
    WriteLine("\n---------------------------------");

    // Вывести содержимое DataTable.
    for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
    {
      for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
```

```
      {
        Write(dt.Rows[curRow][curCol] + "\t");
      }
      WriteLine();
    }
  }
}
```

Вставка данных с помощью сгенерированного кода

Предположим, что теперь нужно вставить новые записи с применением имеющейся строго типизированной объектной модели. Показанный далее вспомогательный метод добавляет две новые строки в текущий объект `InventoryDataTable` и затем обновляет содержимое базы данных через адаптер данных. Первая строка добавляется вручную за счет конфигурирования строго типизированного `DataRow`, а вторая — путем передачи необходимых данных столбцов, что позволяет создать `DataRow` автоматически "за кулисами".

```
public static void AddRecords(
  AutoLotDataSet.InventoryDataTable table,
  InventoryTableAdapter adapter)
{
  try
  {
    // Получить из таблицы новую строго типизированную строку.
    AutoLotDataSet.InventoryRow newRow = table.NewInventoryRow();

    // Заполнить строку данными.
    newRow.Color = "Purple";
    newRow.Make = "BMW";
    newRow.PetName = "Saku";

    // Вставить новую строку.
    table.AddInventoryRow(newRow);

    // Добавить еще одну строку, используя перегруженный метод добавления.
    table.AddInventoryRow("Yugo", "Green", "Zippy");

    // Обновить базу данных.
    adapter.Update(table);
  }
  catch (Exception ex)
  {
    WriteLine(ex.Message);
  }
}
```

Метод `AddRecords()` можно вызвать в `Main()` и таблица базы данных обновится новыми записями:

```
static void Main(string[] args)
{
  ...
  // Добавить строки, обновить и вывести повторно.
  AddRecords(table, adapter);
  table.Clear();
  adapter.Fill(table);
  PrintInventory(table);
  Console.ReadLine();
}
```

Удаление данных с помощью сгенерированного кода

Удаление записей с помощью этой строго типизированной объектной модели также реализуется просто. Автоматически сгенерированный метод `FindBy`*`XXX`*`()` (где *XXX* — имя столбца первичного ключа) строго типизированного класса `DataTable` возвращает корректный (строго типизированный) объект `DataRow`, используя первичный ключ. Взгляните на вспомогательный метод, который удаляет две только что созданных записи:

```
private static void RemoveRecords(
  AutoLotDataSet.InventoryDataTable table, InventoryTableAdapter adapter)
{
  try
  {
    AutoLotDataSet.InventoryRow rowToDelete = table.FindByCarId(1);
    adapter.Delete(rowToDelete.CarId, rowToDelete.Make, rowToDelete.Color,
      rowToDelete.PetName);
    rowToDelete = table.FindByCarId(2);
    adapter.Delete(rowToDelete.CarId, rowToDelete.Make, rowToDelete.Color,
    rowToDelete.PetName);
  }
  catch (Exception ex)
  {
    WriteLine(ex.Message);
  }
}
```

После вызова метода `RemoveRecords()` в `Main()` и повторного вывода содержимого таблицы вы должны заметить, что две ранее вставленных тестовых записи больше не отображаются.

На заметку! При желании пример можно сделать более гибким, запрашивая данные у пользователя с применением класса `Console`.

Вызов хранимой процедуры с помощью сгенерированного кода

Давайте рассмотрим еще один пример использования строго типизированной объектной модели. Мы создадим последний метод, который вызывает хранимую процедуру `GetPetName`. Когда строились адаптеры данных для базы `AutoLot`, был сгенерирован специальный класс по имени `QueriesTableAdapter`, который инкапсулирует процесс вызова хранимых процедур реляционной базы данных. Ниже приведен код финального вспомогательного метода, отображающего название указанного автомобиля:

```
public static void CallStoredProc()
{
  try
  {
    var queriesTableAdapter = new QueriesTableAdapter();
    Write("Enter ID of car to look up: ");
    string carID = ReadLine() ?? "0";
    string carName = "";
    queriesTableAdapter.GetPetName(int.Parse(carID), ref carName);
    WriteLine($"CarID {carID} has the name of {carName}");
  }
```

```
  catch(Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

К настоящему моменту вы знаете, как работать со строго типизированными классами базы данных и упаковывать их в отдельную библиотеку классов. В сгенерированной объектной модели вы обнаружите и другие аспекты, с которыми можно поэкспериментировать, но вам уже известно достаточно, чтобы самостоятельно разобраться в них. В завершение приложения будет показано, как применять запросы LINQ к объекту `DataSet` из ADO.NET.

Исходный код. Проект `StronglyTypedDataSetConsoleClient` доступен в подкаталоге `Appendix_A`.

Программирование с помощью LINQ to DataSet

В этом приложении вы узнали, что данными внутри объекта `DataSet` можно манипулировать тремя различными способами:

- с использованием коллекций `Tables`, `Rows` и `Columns`;
- с применением объектов чтения таблиц данных;
- с использованием строго типизированных классов данных.

Разнообразные индексаторы типов `DataSet` и `DataTable` позволяют взаимодействовать с содержащимися данными в прямолинейной, но слабо типизированной манере. Вспомните, что такой подход требует трактовки данных как табличного блока ячеек, что демонстрируется в следующем примере:

```
static void PrintDataWithIndxers(DataTable dt)
{
  // Вывести содержимое DataTable.
  for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
  {
    for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
      Write(dt.Rows[curRow][curCol + "\t");
    }
    WriteLine();
  }
}
```

Метод `CreateDataReader()` класса `DataTable` предлагает другой подход, при котором данные, содержащиеся в `DataSet`, трактуются как линейный набор строк, предназначенный для обработки последовательным образом. Это позволяет применять к автономному объекту `DataSet` модель программирования с подключенными объектами чтения данных.

```
static void PrintDataWithDataTableReader(DataTable dt)
{
  // Получить объект DataTableReader.
  DataTableReader dtReader = dt.CreateDataReader();
  while (dtReader.Read())
  {
```

```
    for (int i = 0; i < dtReader.FieldCount; i++)
    {
      Write($"{dtReader.GetValue(i)}\t");
    }
    WriteLine();
  }
  dtReader.Close();
}
```

И, наконец, строго типизированный `DataSet` можно использовать для получения кодовой базы, позволяющей взаимодействовать с данными в объекте через свойства, которые отображаются на имена столбцов в реляционной базе данных. Строго типизированные объекты позволяют писать код следующего вида:

```
static void AddRowWithTypedDataSet()
{
  InventoryTableAdapter invDA = new InventoryTableAdapter();
  AutoLotDataSet.InventoryDataTable inv = invDA.GetData();
  inv.AddInventoryRow("Ford", "Yellow", "Sal");
  invDA.Update(inv);
}
```

Хотя все упомянутые подходы имеют свои сценарии применения, API-интерфейс LINQ to DataSet предоставляет еще одну возможность манипулирования данными `DataSet` с использованием выражений запросов LINQ.

На заметку! С помощью API-интерфейса LINQ to DataSet запросы LINQ применяются только к объектам `DataSet`, возвращаемым адаптерами данных, но это никак не связано с применением запросов LINQ напрямую к механизму базы данных. Глава 22 посвящена введению в платформу ADO.NET Entity Framework, которая предлагает способ представления запросов SQL как запросов LINQ.

В первоначальном виде объект `DataSet` из ADO.NET и связанные с ним типы, такие как `DataTable` и `DataView`, не обладают необходимой инфраструктурой, чтобы служить непосредственной целью для запроса LINQ. Например, показанный ниже метод (в котором используются типы из пространства имен `AutoLotDisconnectedLayer`) в результате приводит к ошибке на этапе компиляции:

```
static void LinqOverDataTable()
{
  // Получить объект DataTable с данными.
  InventoryDALDC dal = new InventoryDALDC(
    @"Data Source=(local)\SQLEXPRESS2014;
      Initial Catalog=AutoLot;Integrated Security=True");
  DataTable data = dal.GetAllInventory();

  // Применить запрос LINQ к DataSet?
  var moreData = from c in data where (int)c["CarID"] > 5 select c;
}
```

При попытке компиляции метода `LinqOverDataTable()` компилятор сообщит, что тип `DataTable` предоставляет *реализацию шаблона запросов*. Аналогично процессу применения запросов LINQ к объектам, которые не реализуют интерфейс `IEnumerable<T>`, объекты ADO.NET должны быть трансформированы в совместимый тип. Чтобы понять, как это делается, потребуется исследовать типы из сборки `System.Data.DataSetExtensions.dll`.

Роль библиотеки расширений DataSet

Сборка System.Data.DataSetExtensions.dll, ссылка на которую по умолчанию присутствует во всех проектах Visual Studio, дополняет пространство имен System.Data рядом новых типов (рис. A.28).

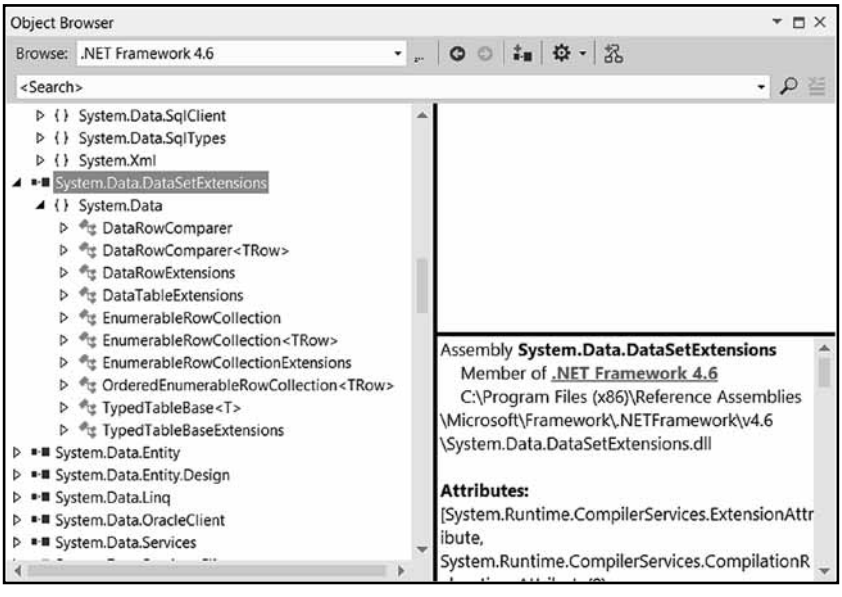

Рис. A.28. Сборка System.Data.DataSetExtensions.dll

Двумя наиболее полезными типами являются классы DataTableExtensions и DataRowExtensions, которые расширяют функциональность типов DataTable и DataRow за счет использования набора расширяющих методов (см. главу 12). Еще один важный класс, TypedTableBaseExtensions, определяет расширяющие методы, которые можно применять к строго типизированным объектам DataSet, чтобы обеспечить поддержку LINQ для внутренних объектов DataTable. Все остальные члены сборки System.Data.DataSetExtensions.dll относятся к чистой инфраструктуре и не предназначены для непосредственного использования в кодовой базе.

Получение объекта DataTable, совместимого с LINQ

А теперь давайте посмотрим, как работать с расширениями DataSet. Предположим, что имеется новый проект консольного приложения C# по имени LinqToDataSetApp. Добавим в него ссылку на последнюю версию (3.0.0.0) сборки AutoLotDAL.dll и модифицируем первоначальный файл кода следующим образом:

```
using System;
...
// Местоположение строго типизированных контейнеров данных.
using AutoLotDAL3.DataSets;

// Местоположение строго типизированных адаптеров данных.
using AutoLotDAL3.DataSets.AutoLotDataSetTableAdapters;
using static System.Console;
```

```
namespace LinqToDataSetApp
{
  class Program
  {
    static void Main(string[] args)
    {
      WriteLine("***** LINQ over DataSet *****\n");
      // Получить строго типизированный объект DataTable, содержащий
      // текущие данные таблицы Inventory из базы данных AutoLot.
      AutoLotDataSet dal = new AutoLotDataSet();
      InventoryTableAdapter tableAdapter = new InventoryTableAdapter();
      AutoLotDataSet.InventoryDataTable data = tableAdapter.GetData();
      // Вызвать описанные ниже методы.
      ReadLine();
    }
  }
}
```

Для преобразования объекта `DataTable` (включая строго типизированный `DataTable`) из ADO.NET в совместимый с LINQ объект потребуется вызвать расширяющий метод `AsEnumerable()`, который определен в классе `DataTableExtensions`. Метод `AsEnumerable()` возвращает объект `EnumerableRowCollection`, содержащий коллекцию объектов `DataRow`.

Затем тип `EnumerableRowCollection` можно использовать для обработки каждой строки с помощью основного синтаксиса `DataRow` (например, синтаксиса индексатора). Рассмотрим показанный ниже новый метод класса `Program`, который принимает строго типизированный `DataTable`, получает перечислимую копию данных и выводит все значения `CarId`:

```
static void PrintAllCarIDs(DataTable data)
{
  // Получить перечислимую версию DataTable.
  EnumerableRowCollection enumData = data.AsEnumerable();
  // Вывести значения идентификаторов автомобилей.
  foreach (DataRow r in enumData)
  {
    WriteLine($"Car ID = {r["CarID"]}");
  }
}
```

Здесь запрос LINQ еще не применялся, но суть в том, что объект `enumData` теперь может служить целью выражения запроса LINQ. Обратите внимание, что объект `EnumerableRowCollection` содержит коллекцию объектов `DataRow`, т.к. для вывода значений столбца `CarId` к каждому подобъекту применяется индексатор типа.

В большинстве случаев нет необходимости объявлять переменную типа `EnumerableRowCollection` для хранения возвращаемого значения `AsEnumerable()`. Взамен метод `AsEnumerable()` можно вызывать внутри самого выражения запроса. Далее приведен код более интересного метода класса `Program`, который получает проекцию `CarId` и `Make` из всех записей в `DataTable`, представляющих автомобили черного цвета (если в вашей таблице `Inventory` нет записей для черных автомобилей, то соответствующим образом измените цвет в запросе LINQ):

```
static void ShowRedCars(DataTable data)
{
  // Спроецировать новый результирующий набор, содержащий
  // идентификатор/цвет для строк, в которых Color = Black.
  var cars = from car in data.AsEnumerable()
```

```
            where
              (string)car["Color"] == "Black"
            select new
            {
              ID = (int)car["CarID"],
              Make = (string)car["Make"]
            };
  WriteLine("Here are the red cars we have in stock:");
  foreach (var item in cars)
  {
    WriteLine($"-> CarID = {item.ID} is {item.Make}");
  }
}
```

Роль расширяющего метода DataRowExtensions.Field<T>()

Один из нежелательных аспектов текущего выражения запроса LINQ связан с тем, что для получения результирующего набора используются многочисленные операции приведения и индексаторы `DataRow`. Это может привести к генерации исключений времени выполнения, если будет предпринята попытка приведения к несовместимому типу данных. Чтобы привнести в запрос строгую типизацию, можно применить расширяющий метод `Field<T>()` типа `DataRow`. Такой прием позволяет увеличить безопасность к типам запроса, поскольку совместимость типов данных проверяется на этапе компиляции. Взгляните на следующее изменение:

```
var cars = from car in data.AsEnumerable()
  where
    car.Field<string>("Color") == "Black"
  select new
  {
    ID = car.Field<int>("CarID"),
    Make = car.Field<string>("Make")
  };
```

В этом случае можно вызвать метод `Field<T>()` и указать параметр типа для представления лежащего в основе типа данных столбца. В качестве аргумента методу передается имя столбца. Учитывая дополнительную проверку на этапе компиляции, при обработке элементов `EnumerableRowCollection` рекомендуется использовать метод `Field<T>()`, а не индексатор `DataRow`.

Помимо факта вызова метода `AsEnumerable()` общий формат запроса LINQ идентичен тому, что вы уже видели в главе 13, повторять здесь детали разнообразных операций LINQ не имеет смысла. Дополнительные примеры можно найти в разделе “LINQ to DataSet Examples” (“Примеры LINQ to DataSet”) документации .NET Framework 4.7 SDK.

Заполнение новых объектов DataTable из запросов LINQ

Заполнить данными новый объект `DataTable` можно также на основе результатов запроса LINQ при условии, что в нем *не* применяются проекции. Если есть результирующий набор, тип которого может быть представлен как `IEnumerable<T>`, тогда на нем можно вызвать расширяющий метод `CopyToDataTable<T>()`, как показано ниже:

```
static void BuildDataTableFromQuery(DataTable data)
{
  var cars = from car in data.AsEnumerable()
             where car.Field<int>("CarID") > 5
             select car;
```

```
//Использовать этот результирующий набор для построения нового объекта DataTable.
DataTable newTable = cars.CopyToDataTable();

// Вывести содержимое DataTable.
for (int curRow = 0; curRow < newTable.Rows.Count; curRow++)
{
  for (int curCol = 0; curCol < newTable.Columns.Count; curCol++)
  {
    Write(newTable.Rows[curRow][curCol].ToString().Trim() + "\t");
  }
  WriteLine();
}
}
```

На заметку! С использованием расширяющего метода `AsDataView<T>()` запрос LINQ можно также трансформировать в тип `DataView`.

Продемонстрированный прием может оказаться удобным, когда результат запроса LINQ нужно задействовать в качестве источника для операции привязки к данным. Вспомните, что элемент управления `DataGridView` в Windows Forms (а также элемент управления типа сетки в ASP.NET или WPF) поддерживает свойство по имени `DataSource`. Привязать результат запроса LINQ к сетке можно было бы следующим образом:

```
// Предположим, что myDataGrid - объект сетки графического
// пользовательского интерфейса.
myDataGrid.DataSource = (from car in data.AsEnumerable()
                         where car.Field<int>("CarID") > 5
                         select car).CopyToDataTable();
```

Итак, исследование автономного уровня ADO.NET завершено. С применением этого аспекта API-интерфейса можно извлекать данные из реляционной базы, обрабатывать их и возвращать обратно в базу, удерживая подключение к базе данных открытым на протяжении минимально возможного промежутка времени.

Исходный код. Проект `LinqToDataSetApp` доступен в подкаталоге `Appendix_A`.

Резюме

В настоящем приложении подробно рассматривался автономный уровень ADO.NET. Как вы видели, центральной частью автономного уровня является тип `DataSet` — размещаемое в памяти представление любого числа таблиц и дополнительно любого количества отношений, ограничений и выражений. Преимущество установления отношений между локальными таблицами связано с тем, что между ними можно программно перемещаться без подключения к удаленному хранилищу данных.

В приложении также была исследована роль типа адаптера данных. С использованием этого типа (а также его свойств `SelectCommand`, `InsertCommand`, `UpdateCommand` и `DeleteCommand`) адаптер может переносить изменения из `DataSet` в исходное хранилище данных. Кроме того, вы научились осуществлять навигацию по объектной модели `DataSet` прямолинейным ручным способом и с помощью строго типизированных объектов, которые обычно генерируют инструменты визуального конструктора наборов данных среды Visual Studio.

Наконец, вы взглянули на один из аспектов набора технологий LINQ под названием LINQ to DataSet. Он позволяет получать поддерживающую запросы копию `DataSet`, которую могут принимать правильно сформированные запросы LINQ.