

Привязка данных

Эта глава основана на материалах главы 26, в которой описаны различные способы извлечения и изменения данных, и посвящена представлению данных для пользователей с привязкой данных к различным элементам управления Windows. Точнее говоря, здесь будут обсуждаться следующие вопросы:

- ☐ отображение данных с помощью элемента управления `DataGridView`;
- ☐ возможности привязки данных .NET и как они работают;
- ☐ как использовать проводник по серверу `Server Explorer` для создания соединения и генерации класса `DataSet` (не написав ни единой строки кода);
- ☐ как использовать проверку попаданий и рефлексии строк `DataGrid`.

Коды примеров этой главы доступны на прилагаемом компакт-диске.

Элемент управления `DataGridView`

Элемент управления `DataGrid`, доступный начиная с ранних выпусков .NET, был вполне функциональным, но имел множество недоработок, которые делали его неподходящим для применения в коммерческих приложениях, например, отсутствовала возможность вывода графических изображений и раскрывающихся элементов, блокировка столбцов и тому подобное. Этот элемент управления был не вполне полноценным, поэтому многие независимые поставщики предлагали собственные сеточные компоненты, которые компенсировали эти недостатки и представляли гораздо более широкую функциональность.

В .NET 2.0 появился дополнительный сеточный элемент управления — `DataGridView`. Он восполнил многие недостатки своего предшественника и добавил важную функциональность, которая до этого была реализована лишь в продуктах независимых поставщиков.

Этот элемент оснащен такими же средствами привязки данных, как и старый `DataGrid`, а потому может работать совместно с классами `Array`, `DataTable`, `DataView` или `DataSet` либо компонентами, реализующими интерфейс `IListSource` или `IList`. Элемент управления `DataGridView` обеспечивает возможности разнообразного представления одних и тех же данных. В простейшем случае отображаемые данные (такие как из `DataSet`) указываются установкой значений свойств `DataSource` и `DataMember`. Отметим, что этот элемент управления не может подставляться вместо `DataGrid`, потому что его программный интерфейс полностью отличается от интерфейса `DataGrid`. К тому же он предлагает более широкие возможности, о которых мы и поговорим в этой главе.

Отображение табличных данных

В главе 19 было представлено множество способов выбора данных и чтения их в таблицы, хотя отображались они очень примитивным способом — с использованием `Console.WriteLine()`.

Следующий пример демонстрирует, как извлечь данные и отобразить их в элементе управления `DataGridView`. Для этой цели мы создадим новое приложение `DisplayTabularData`, внешний вид которого показан на рис. 32.1.

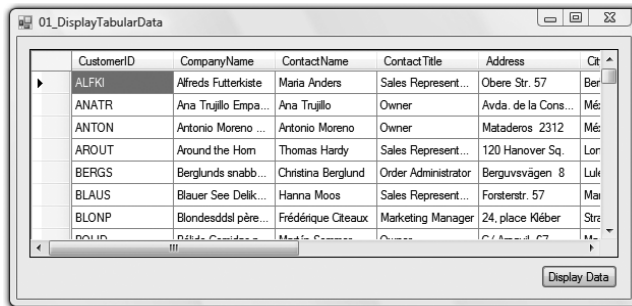


Рис. 32.1. Внешний вид приложения `DisplayTabularData`

Это простое приложение выбирает каждую запись из таблицы `Customer` базы данных `Northwind` и отображает эти записи пользователю в элементе управления `DataGridView`. Ниже показан код этого примера (исключая код определения формы и элемента управления).

```
using System;
using System.Configuration;
using System.Data;
using System.Data.Common;
using System.Data.SqlClient;
using System.Windows.Forms;
namespace DisplayTabularData
{
    partial class Form1: Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void getData_Click(object sender, EventArgs e)
        {
            string customers = "SELECT * FROM Customers";
            using (SqlConnection con = new SqlConnection (ConfigurationManager.
                ConnectionStrings["northwind"].ConnectionString))
            {
                DataSet ds = new DataSet();
                SqlDataAdapter da = new SqlDataAdapter(customers, con);
                da.Fill(ds, "Customers");
                dataGridView.AutoGenerateColumns = true;
                dataGridView.DataSource = ds;
                dataGridView.DataMember = "Customers";
            }
        }
    }
}
```

Форма состоит из кнопки `getData`, в результате щелчка на которой вызывается метод `getData_Click()`, показанный в коде примера.

При этом конструируется объект `SqlConnection`, использующий свойство `ConnectionStrings` класса `ConfigurationManager`. Далее создается набор данных и заполняется на основе таблицы базы данных с помощью объекта `DataAdapter`. Затем эти данные отображаются элементом управления `DataGridView` за счет установки свойств `DataSource` и `DataMember`.

Отметим, что свойству `AutoGenerateColumns` также присваивается значение `true`, поскольку это гарантирует, что пользователь что-то увидит. Если этот флаг не установлен, все столбцы придется создавать самостоятельно.

Источники данных

Элемент управления `DataGridView` предлагает гибкий способ отображения данных; в дополнение к установке `DataSource` равным `DataSet`, а `DataMember` — равным имени отображаемой таблицы, свойство `DataSource` может указывать на любой из следующих источников:

- ☐ массив (визуальная таблица может быть связана с любым одномерным массивом);
- ☐ `DataTable`;
- ☐ `DataView`;
- ☐ `DataSet` или `DataViewManager`;
- ☐ компоненты, реализующие интерфейс `IListSource`;
- ☐ компоненты, реализующие интерфейс `IList`;
- ☐ любой обобщенный класс коллекции или объект, унаследованный от обобщенного класса коллекции.

В последующих разделах будут представлены примеры применения каждого из упомянутых источников данных.

Отображение данных из массива

На первый взгляд это кажется простым. Нужно создать массив, наполнить его некоторыми данными и установить свойство `DataSource` элемента управления `DataGridView`. Вот пример кода:

```
string[] stuff = new string[] { "One", "Two", "Three" };
dataGridView.DataSource = stuff;
```

Если источник данных включает множество возможных таблиц-кандидатов (как в случае с `DataSet` или `DataViewManager`), также понадобится установить свойство `DataMember`.

Можно заменить этим кодом код события `getData_Click` из предыдущего примера. Проблема с этим кодом проявляется в отображении данных (рис. 32.2).

Вместо отображения строк, определенных в массиве, таблица показывает длины этих строк. Причина состоит в том, что когда массив используется в качестве источника данных для `DataGridView`, то он ищет первое общедоступное свойство объекта, содержащегося в массиве, вместо строкового значения. Первое (и единственное) общедоступное свойство строки —

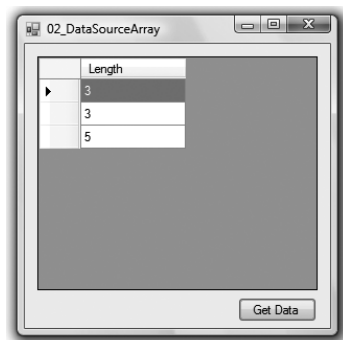


Рис. 32.2. Проблема, связанная с примером кода

это длина, потому она и отображается. Список свойств каждого класса может быть получен с помощью метода `GetProperties` класса `TypeDescriptor`. Он возвращает коллекцию объектов `PropertyDescriptor`, которая затем может использоваться для отображения данных. Элемент управления .NET `PropertyGrid` применяет этот метод для отображения произвольных объектов.

Один из способов решения этой проблемы с отображением строк в `DataGridView` заключается в создании класса-оболочки:

```
protected class Item
{
    public Item(string text)
    {
        _text = text;
    }
    public string Text
    {
        get{return _text;}
    }
    private string _text;
}
```

На рис. 32.3 показан вывод массива объектов класса `Item` (который может быть и структурой), служащего источником данных.



Рис. 32.3. Вывод массива объектов класса `Item`

DataTable

`DataTable` можно отображать в элементе управления `DataGridView` двумя способами:

- ❑ если используется `DataTable` сама по себе, следует просто установить свойство `DataSource` элемента управления на эту таблицу;
- ❑ если `DataTable` входит в `DataSet`, необходимо установить `DataSource` на `DataSet`, а свойство `DataMember` — равным имени `DataTable` внутри этого `DataSet`.

На рис. 32.4 показан результат запуска кода примера `DataSourceDataTable`.

Обратите внимание на отображение последнего столбца — он показывает флажки вместо более привычных полей редактирования. Дело в том, что элемент управления `DataGridView` в отсутствие любой другой информации будет читать схему из источника данных (в данном случае — таблицы `Products`) и на основе типа столбца автоматически выбирать тип элемента управления, отображающего его. В отличие от `DataGrid`, `DataGridView` имеет встроенную поддержку столбцов с изображениями, кнопками и комбинированными списками.

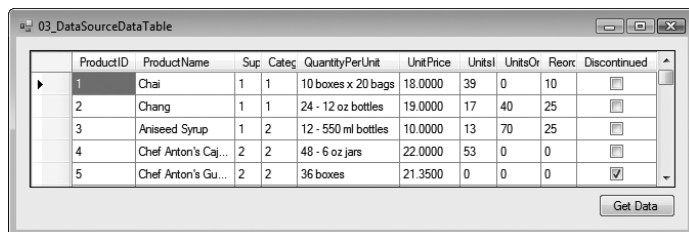


Рис. 32.4. Результат запуска кода примера `DataSourceDataTable`

Данные в базе не изменяются, когда изменяются поля в визуальной таблице, поскольку ее данные хранятся локально на клиентском компьютере — нет никакого активного подключения к базе данных. Обновление данных в базе мы обсудим в этой главе позднее.

Отображение данных из *DataGridView*

DataGridView предоставляет средства фильтрации и сортировки данных внутри *DataTable*. Когда данные выбираются из базы, обычно пользователю разрешено сортировать их, например, щелчком на заголовке того или иного столбца. В дополнение пользователь может пожелать отфильтровать данные, чтобы показать только определенные строки, например, те, что подверглись изменениям. *DataGridView* можно фильтровать так, что только выбранные строки будут показаны пользователю; однако это не ограничивает столбцы из *DataTable*.

DataGridView не позволяет фильтровать столбцы, а только строки.

Чтобы создать *DataGridView* на базе существующей *DataTable*, нужно использовать следующий код:

```
DataGridView dv = new DataGridView(dataTable);
```

После создания *DataGridView* его настройки могут быть изменены. Это касается данных и допустимых действий над ними, когда они отображаются в табличном виде, например:

- ☐ установка `AllowEdit = false` отключает всю функциональность редактирования столбцов в строках;
- ☐ установка `AllowNew = false` отключает функциональность новых строк;
- ☐ установка `AllowDelete = false` отключает возможность удаления строк;
- ☐ установка `RowStateFilter` отображает только строки с заданным состоянием;
- ☐ установка `RowFilter` позволяет фильтровать строки на основе выражений.

В следующем разделе будет показано, как использовать настройки `RowFilter` и `RowStateFilter`; остальные настройки объяснений не требуют.

Фильтрация строк на основе данных

После того, как создан *DataGridView*, данные, отображенные в этом представлении, могут быть изменены установкой свойства `RowFilter`. Это свойство, задаваемое строкой, применяется в качестве средства фильтрации на базе критериев, определенных значением строки. Ее синтаксис подобен конструкции `WHERE` стандартного SQL, но относится к данным, уже полученным из базы.

В табл. 32.1 показаны некоторые примеры конструкций фильтрации.

Исполняющая система старается наилучшим образом подогнать типы данных в выражениях фильтра к соответствующим типам исходных столбцов. Например, вполне допустимо было в предыдущем примере написать `UnitsInStock > '50'` несмотря на то, что столбец имеет целочисленный тип. Если указывается неправильное выражение фильтра, это приводит к возбуждению исключения `EvaluateException`.

Таблица 32.1. Примеры конструкций фильтрации

Конструкция	Описание
<code>UnitsInStock > 50</code>	Показывает только те строки, в которых значение столбца <code>UnitsInStock</code> больше 50.
<code>Client = 'Smith'</code>	Возвращает записи для заданного клиента.
<code>County LIKE 'C*'</code>	Возвращает все записи, в которых поле <code>Country</code> начинается с <code>C</code> — в данном примере будут возвращены строки со следующими значениями <code>Country</code> : <code>Cornwall</code> , <code>Cumbria</code> , <code>Cheshire</code> и <code>Cambridgeshire</code> . Символ <code>%</code> может использоваться как односимвольный шаблон, в то время как <code>*</code> — обобщенный шаблон, которому соответствует ноль или более символов.

Фильтрация на основе состояния строк

Каждая строка внутри `DataView` имеет определенное состояние, которое может принимать одно из значений, перечисленных в табл. 32.2. Это состояние также может использоваться для фильтрации строк, видимых пользователю.

Таблица 32.2. Допустимые состояния строк

<code>DataViewRowState</code>	Описание
<code>Added</code>	Список всех вставленных строк.
<code>CurrentRows</code>	Список всех строк за исключением удаленных.
<code>Deleted</code>	Список всех исходных строк, которые были выделены и удалены; вновь созданные и тут же удаленные строки не отображаются.
<code>ModifiedCurrent</code>	Список всех измененных строк с их текущими значениями.
<code>ModifiedOriginal</code>	Список всех измененных строк в их исходном состоянии (до изменения).
<code>OriginalRows</code>	Список всех строк, которые были изначально получены от источника данных. Не включает новые строки. Показывает исходные значения столбцов (то есть те, что были до внесения изменений).
<code>Unchanged</code>	Список всех строк, которые не были изменены.

На рис. 32.5 показана одна таблица, которая может содержать добавленные, удаленные или измененные строки, и вторая таблица, включающая строки с одним из перечисленных состояний.

Фильтр не только применяется к видимым строкам, но также и к столбцам внутри этих строк. Это наглядно демонстрирует выбор `ModifiedOriginal` или `ModifiedCurrent`. Доступные состояния описаны в главе 20 и основаны на перечислении `DataRowVersion`.

Например, когда пользователь обновляет столбец в строке, то эта строка будет отображена как при выборе `ModifiedOriginal`, так и при `ModifiedCurrent`; однако действительным значением будет либо исходное значение `Original`, полученное из базы данных (если выбрано `ModifiedOriginal`), либо текущее измененное значение `DataRowVersion` (в случае выбора `ModifiedCurrent`).

Сортировка строк

Помимо фильтрации, данные в `DataView` можно также сортировать. Сортировка выполняется по возрастанию или по убыванию — простым щелчком на заголовке столбца в элементе управления `DataGridView` (рис. 32.6).

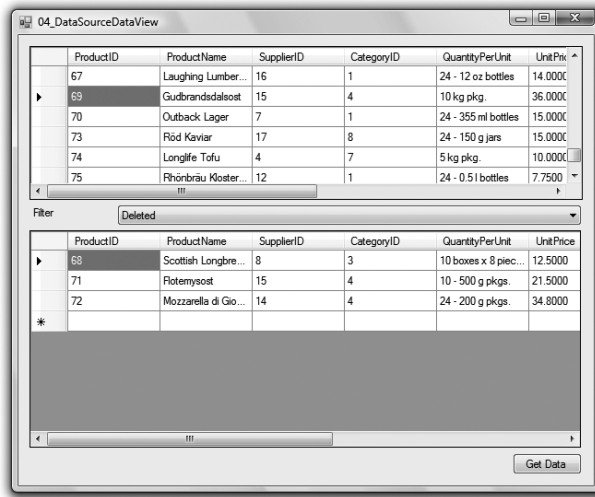


Рис. 32.5. Пример с двумя таблицами

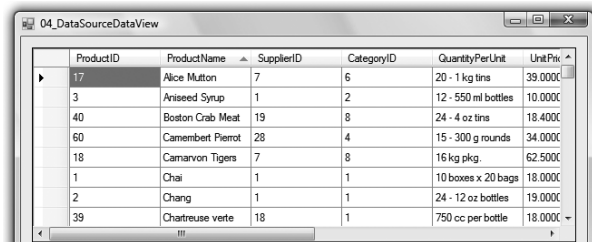


Рис. 32.6. Сортировка с использованием элемента управления DataGridView

Единственная проблема состоит в том, что сортировать можно только по одному столбцу, в то время как лежащий в основе элемент управления DataView может сортировать по множеству столбцов. Когда столбец сортируется — либо щелчком на его заголовке (как показано, на столбце ProductName), либо программно — DataGridView показывает в его заголовке изображение стрелочки для пометки, по какому столбцу выполнена сортировка.

Программная установка порядка сортировки реализуется путем присвоения свойству Sort в DataView соответствующего значения:

```
dataView.Sort = "ProductName";
dataView.Sort = "ProductName ASC, ProductID DESC";
```

Первая строка сортирует данные по значению столбца ProductName, что можно видеть на рис. 32.6. Вторая строка сортирует данные в возрастающем порядке по ProductName и в убывающем — по ProductID.

DataView поддерживает сортировку и по возрастанию значений столбца, и по их убыванию. Если данные в DataView отсортированы по более чем одному столбцу, то DataGridView перестает показывать стрелочки-символы сортировки в заголовках.

Каждый столбец в визуальной таблице может быть строго типизированным, поэтому его порядок сортировки будет основан не на строковом представлении данных столбца, а на самих данных. В результате, если в DataGridView присутствует столбец даты, то пользователь может отсортировать строки по датам, а не по их строковым представлениям (которое может иметь различный формат).

Отображение данных из класса DataSet

Существует одно средство DataSet, которое отличает DataGridView от DataGrid — это когда DataSet определяет отношение между таблицами. Как в приведенных выше примерах с DataGridView, DataGrid может одновременно отображать только одну таблицу. Однако, как показано в следующем примере с DataSourceDataSet, можно выполнять навигацию по отношениям внутри DataSet в пределах одного экрана. Следующий код можно использовать для генерации такого DataSet на основе таблиц Customers и Orders базы данных Northwind. Этот пример загружает данные из этих двух DataTable и затем создает отношение между ними под названием CustomerOrders:

```
string orders = "SELECT * FROM Orders";
string customers = "SELECT * FROM Customers";
SqlConnection conn = new SqlConnection(source);
SqlDataAdapter da = new SqlDataAdapter(orders, conn);
DataSet ds = new DataSet();
da.Fill(ds, "Orders");
da = new SqlDataAdapter(customers, conn);
da.Fill(ds, "Customers");
ds.Relations.Add("CustomerOrders",
    ds.Tables["Customers"].Columns["CustomerID"],
    ds.Tables["Orders"].Columns["CustomerID"]);
```

Созданный DataSet связывается с DataGrid простым вызовом SetDataBinding():

```
dataGrid1.SetDataBinding(ds, "Customers");
```

Это даст вывод, показанный на рис. 32.7.

В отличие от других примеров DataGridView, приведенных с этой главе, здесь мы видим знак + слева от каждой записи. Это отражает тот факт, что DataSet имеет управляемое отношение между заказчиками и заказами. В коде можно определить любое количество таких отношений.

Когда пользователь щелкает на знаке +, отображается список отношений (или скрывается — если ранее он был отображен). Щелчок на имени отношения позволяет перейти к связанным записям (рис. 32.8); в данном примере — к списку заказов, размещенных выбранным заказчиком.

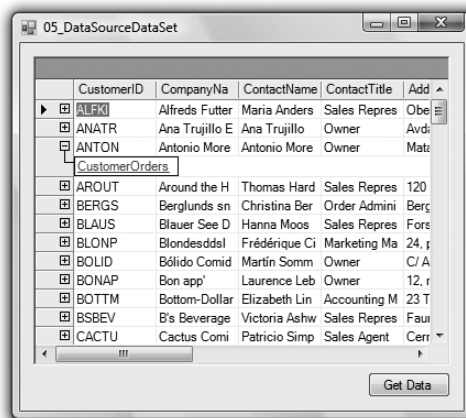


Рис. 32.7. Знаки + слева от каждой записи обозначают отношения между заказчиками и заказами

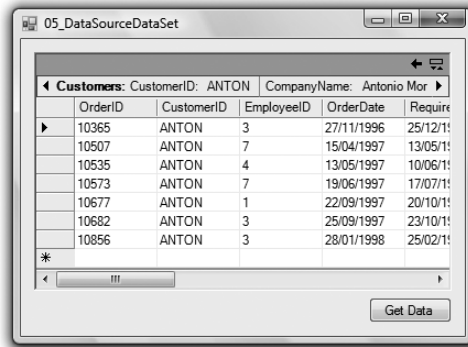


Рис. 32.8. Связанные записи, отображаемые в результате щелчка на имени отношения

Элемент управления DataGridView также включает пару новых пиктограмм в правом верхнем углу. Пиктограмма со стрелкой влево позволяет перейти к родительской записи и вернет экран к состоянию, показанному на предыдущей странице. Вторая пиктограмма служит для того, чтобы показать/скрыть детальную информацию о родительской записи в строке заголовка.

Отображение данных в DataGridView

Отображение данных в DataGridView — такое же, как для DataSet, показанное в предыдущем разделе. И потому для отображения только заказчиков из Великобритании (UK), содержащихся в таблице Customers, необходим следующий код:

```
DataGridView dvm = new DataGridView(ds);
dvm.DataViewSettings["Customers"].RowFilter = "Country='UK'";
dataGridView1.SetDataBinding(dvm, "Customers");
```

На рис. 32.9 показан вывод примера кода DataGridViewDataSourceViewManager.

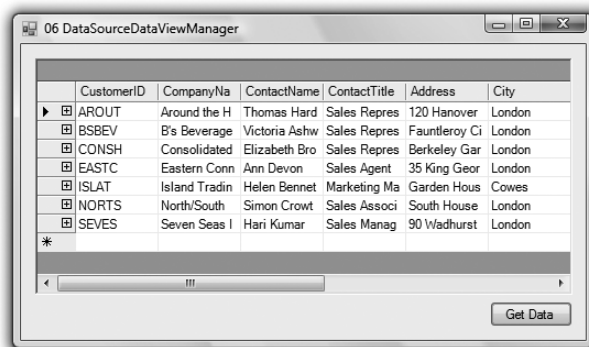


Рис. 32.9. Вывод примера кода DataGridViewDataSourceViewManager

Интерфейсы IListSource и IList

DataGridView также поддерживает любые объекты, которые реализуют интерфейсы IListSource или IList. Интерфейс IListSource имеет только один метод — GetList(), который возвращает интерфейс IList. В свою очередь, IList пред-

ставляет собой нечто более интересное. Этот интерфейс реализуют многие классы исполняющей системы. Среди них — `Array`, `ArrayList` и `StringCollection`.

Когда используется `ICollection`, в силе остается то же предупреждение относительно объектов, содержащихся в коллекции, которое было упомянуто ранее в примере с `Array` в качестве источника данных, а именно — если источником данных `DataGrid` служит `StringCollection`, то в визуальной таблице отображаются длины строк, а не их текст, как ожидалось.

Отображение обобщенных коллекций

В дополнение к уже описанным типам, `DataGridView` также поддерживает привязку к обобщенным коллекциям. При этом применяется такой же синтаксис, как и в других примерах, приведенных ранее в этой главе — нужно просто установить свойство `DataSource` в ссылку на коллекцию, и элемент управления сгенерирует соответствующий экран.

Опять-таки, отображаемые столбцы основаны на свойствах объекта — все общедоступные читаемые поля отображаются в `DataGridView`. В следующем примере демонстрируется отображение списочного класса.

```
class PersonList : List < Person >
{
}
class Person
{
    public Person( string name, Sex sex, DateTime dob )
    {
        _name = name;
        _sex = sex;
        _dateOfBirth = dob;
    }
    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
    public Sex Sex
    {
        get { return _sex; }
        set { _sex = value; }
    }
    public DateTime DateOfBirth
    {
        get { return _dateOfBirth; }
        set { _dateOfBirth = value; }
    }
    private string _name;
    private Sex _sex;
    private DateTime _dateOfBirth;
}
enum Sex
{
    Male,
    Female
}
```

Экран показывает несколько экземпляров класса `Person`, которые сконструированы внутри класса `PersonList` (рис. 32.10).

Элемент управления при отображении данных использует объекты, унаследованные от `DataGridViewColumn` — и как можно видеть на рис. 32.11, теперь доступно намного больше опций отображения данных, чем было раньше в оригинальном `DataGrid`. Основной недостаток `DataGrid` — отсутствие возможности отображения столбцов с раскрывающимися списками — теперь эта функциональность реализована в `DataGridView` в виде `DataGridViewComboBoxColumn`.

Когда указывается источник данных для `DataGridView`, по умолчанию он конструирует свои столбцы автоматически. Столбцы создаются на основе типов данных источника, поэтому, например, любое поле булевского типа отображается на `DataGridViewCheckBoxColumn`. Если вы предпочитаете создавать столбцы самостоятельно, то для этого можно установить значение `false` свойству `AutoGenerateColumns` и сконструировать столбцы самостоятельно.

В следующем примере показано, как можно определить столбцы с включением в них графических изображений, а также столбцы с раскрывающимися списками. Код использует `DataSet` и извлекает данные в два объекта `DataTable`. Первая таблица `DataTable` содержит информацию о наемных работниках из базы данных `Northwind`. Вторая таблица состоит из столбца `EmployID` и сгенерированного столбца `Name`, используемого при отображении раскрывающегося списка.

```
using (SqlConnection con =
    new SqlConnection (
        ConfigurationSettings.ConnectionStrings["northwind"].ConnectionString ) )
{
    string select = "SELECT EmployeeID, FirstName, LastName, Photo,
                    IsNull(ReportsTo,0) as ReportsTo FROM Employees";
    SqlDataAdapter da = new SqlDataAdapter(select, con);
    DataSet ds = new DataSet();
    da.Fill(ds, "Employees");
    select = "SELECT EmployeeID, FirstName + ' ' + LastName as Name
            FROM Employees UNION SELECT 0, '(None) '";
    da = new SqlDataAdapter(select, con);
    da.Fill(ds, "Managers");

    // Сконструировать столбцы для табличного экранного представления
    SetupColumns(ds);
    // Высота строки по умолчанию
    dataGridView.RowTemplate.Height = 100;

    // Настроить источник данных
    dataGridView.AutoGenerateColumns = false;
    dataGridView.DataSource = ds.Tables["Employees"];
}
```

Здесь следует отметить две вещи. Первое — оператор `select` заменяет значения `NULL` в столбце `ReportsTo` нулевыми значениями. В базе данных есть одна запись, содержащая в этом поле `NULL`, что говорит о том, что данный сотрудник не имеет руководителя. Однако когда данные связываются, `ComboBox` требует значения в этом столбце, иначе при отображении таблицы будет возбуждено исключение. В данном примере выбрано значение ноль, потому что оно отсутствует в таблице — обычно это называется *сигнальным значением*, поскольку имеет в приложении специальный смысл.

Вторая конструкция SQL извлекает данные для `ComboBox` и включает искусственную строку со значениями `Zero` и `(None)`. На рис. 32.12 вторая строка отображает значение `(None)`.

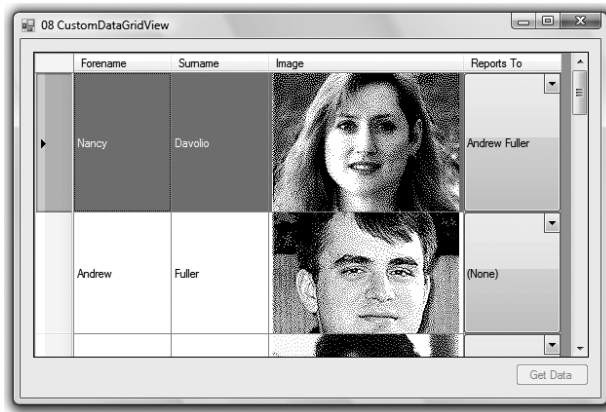


Рис. 32.12. Отображение значения (None)

Пользовательские столбцы создаются следующей функцией:

```
private void SetupColumns(DataSet ds)
{
    DataGridViewTextBoxColumn forenameColumn = new DataGridViewTextBoxColumn();
    forenameColumn.DataPropertyName = "FirstName";
    forenameColumn.HeaderText = "Forename";
    forenameColumn.ValueType = typeof(string);
    forenameColumn.Frozen = true;
    dataGridView.Columns.Add(forenameColumn);

    DataGridViewTextBoxColumn surnameColumn = new DataGridViewTextBoxColumn();
    surnameColumn.DataPropertyName = "LastName";
    surnameColumn.HeaderText = "Surname";
    surnameColumn.Frozen = true;
    surnameColumn.ValueType = typeof(string);
    dataGridView.Columns.Add(surnameColumn);

    DataGridViewImageColumn photoColumn = new DataGridViewImageColumn();
    photoColumn.DataPropertyName = "Photo";
    photoColumn.Width = 100;
    photoColumn.HeaderText = "Image";
    photoColumn.ReadOnly = true;
    photoColumn.ImageLayout = DataGridViewImageCellLayout.Normal;
    dataGridView.Columns.Add(photoColumn);

    DataGridViewComboBoxColumn reportsToColumn = new DataGridViewComboBoxColumn();
    reportsToColumn.HeaderText = "Reports To";
    reportsToColumn.DataSource = ds.Tables["Managers"];
    reportsToColumn.DisplayMember = "Name";
    reportsToColumn.ValueMember = "EmployeeID";
    reportsToColumn.DataPropertyName = "ReportsTo";
    dataGridView.Columns.Add(reportsToColumn);
}
```

В этом примере раскрывающийся список (ComboBox) создается последним и используется в качестве источника данных таблицу Managers. Она содержит столбцы Name и EmployeeID и они присваиваются, соответственно, свойствам DisplayMember и ValueMember. Эти свойства определяют, откуда поступают данные для ComboBox.

Свойство `DataPropertyName` устанавливается на столбец в главной таблице данных, к которой привязан раскрывающийся список — это обеспечивает начальное значение столбца, и если пользователь выберет в раскрывающемся списке другую позицию, то это значение будет обновлено.

Единственное, с чем еще необходимо справиться в этом примере — корректная обработка `null`-значений при обновлении базы данных. В данный момент, если в раскрывающемся списке выбрать позицию (`None`), он будет пытаться записать в базу `null`-значение. Это может вызвать исключение со стороны SQL Server, поскольку нарушает ограничение внешнего ключа для столбца `ReportsTo`. Чтобы преодолеть это, потребуется выполнить предварительную обработку данных перед отправкой их SQL Server, установив `NULL` в столбце `ReportsTo` для всех записей, где было значение 0.

Привязка данных

В предшествующих примерах использовались элементы управления `DataGrid` и `DataGridView`, которые составляют лишь небольшую часть из всех элементов управления .NET, применяемых для отображения данных. Процесс связывания элемента управления с источником данных называется *привязкой данных* (data binding).

В библиотеке MFC (Microsoft Foundation Class) процесс связывания данных из переменных класса с набором элементов управления назывался DDX (Dialog Data Exchange — диалоговый обмен данными). Средства связывания данных с элементами управления, доступные в .NET, существенно проще в использовании и к тому же имеют более широкие возможности. Например, в .NET можно связать данные с большинством свойств элемента управления, а не только с отображаемым текстом. Можно также связывать данные в манере, подобной ASP.NET (см. главу 37).

Простая привязка

Элемент управления, поддерживающий простую привязку, обычно отображает одно значение за раз, как это делает текстовое поле или переключатель. В следующем примере показано, как связать столбец из `DataSet` с `TextBox`:

```
DataSet ds = CreateDataSet();
textBox.DataBindings.Add("Text", ds, "Products.ProductName");
```

После извлечения некоторых данных из таблицы `Products` и сохранения их в возвращаемом методом `CreateDataSet()` экземпляре `DataSet` вторая строка привязывает свойство `Text` элемента управления (`textBox`) к столбцу `Products.ProductName`. На рис. 32.13 показан результат привязки данных подобного рода.

Текстовое поле отображает строку из базы данных. На рис. 32.14 демонстрируется использование инструмента SQL Server Management Studio для сверки содержимого таблицы `Products` с отображенным в текстовом поле значением столбца.

Иметь на экране единственное текстовое поле без возможности прокрутки к следующей или предыдущей записи и без возможности обновления данных в базе не очень-то полезно. В следующем разделе мы рассмотрим более реалистичный пример, а также представим другие объекты, необходимые для того, чтобы привязка данных работала.

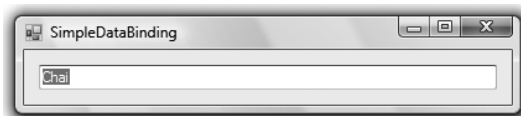


Рис. 32.13. Пример привязки данных

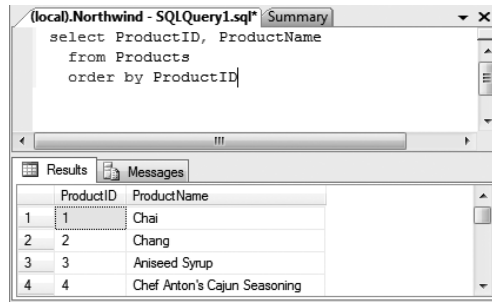


Рис. 32.14. Использование инструмента SQL Server Management Studio для целей верификации

Объекты привязки данных

На рис. 32.15 показана иерархия классов объектов, обеспечивающих привязку данных. В этом разделе мы рассмотрим классы `BindingContext`, `CurrencyManager` и `PropertyManager` пространства имен `System.Windows.Forms` и продемонстрируем, как они взаимодействуют с данными, привязанными к одному или более элементам управления формы.

Объекты, используемые для привязки, выделены серым.

В предыдущем примере свойство `DataBindings` элемента управления `TextBox` применялось для привязки столбца из `DataSet` к свойству `Text` элемента управления. Свойство `DataBindings` — это экземпляр коллекции `ControlBindingsCollection`, показанной на рис. 32.15:

```
textBox1.DataBindings.Add("Text", ds,
    "Products.ProductName");
```

Эта строка привязывает объект `Binding` к `ControlBindingsCollection`.

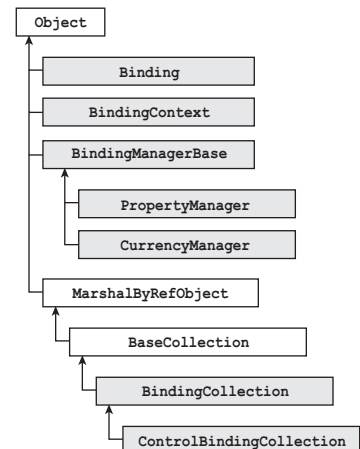


Рис. 32.15. Иерархия классов объектов, обеспечивающих привязку данных

BindingContext

Каждая форма `Windows Forms` имеет свойство `BindingContext`. Кстати, класс `Form` унаследован от `Control`, в котором это свойство на самом деле определено, потому большинство элементов управления обладают этим свойством. Объект `BindingContext` включает коллекцию экземпляров `BindingManagerBase` (рис. 32.16). Эти экземпляры создаются и добавляются к объекту диспетчера привязки в момент связывания элемента управления с данными.

`BindingContext` может включать несколько источников данных, помещенных в оболочку либо `CurrencyManager`, либо `PropertyManager`. Решение о том, какой именно класс использовать, основывается на самом источнике данных.

Если источник данных включает список элементов, такой как `DataTable`, `DataGridView`, или же любой объект, реализующий интерфейс `IList`, применяется `CurrencyManager`. Объект `CurrencyManager` может поддерживать текущую позицию внутри источника данных. Если же источник данных возвращает только одно значение, в `BindingContext` помещается `PropertyManager`.

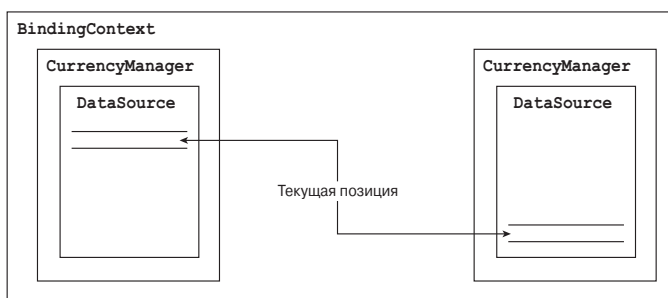


Рис. 32.16. Объект *BindingContext* включает коллекцию экземпляров *BindingManagerBase*

Для конкретного источника данных *CurrencyManager* или *PropertyManager* создается лишь однажды. Если два текстовых поля привязаны к строке *DataTable*, в контексте привязки будет создан только один *CurrencyManager*.

Каждый элемент управления, добавленный к форме, связывается с диспетчером привязки формы, так что все элементы управления разделяют один и тот же его экземпляр. Когда элемент управления создается, его свойство *BindingContext* равно *null*. Когда же он добавляется к коллекции *Controls* формы, его свойство *BindingContext* устанавливается равным *BindingContext* формы.

Чтобы привязать элемент управления к форме, к его свойству *BindingContext* должен добавиться элемент, представляющий собой экземпляр *ControlBindingsCollection*. Следующий код создает новую привязку:

```
textBox.DataBindings.Add("Text", ds, "Products.ProductName");
```

Внутри метод *Add()* объекта *ControlBindingsCollection* создает новый экземпляр объекта *Binding* на основании параметра, переданного методу, и добавляет его к коллекции привязок, представленной на рис. 32.17.

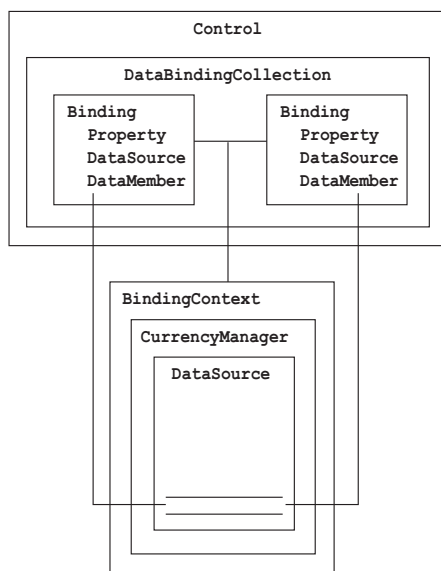


Рис. 32.17. Добавление к *Control* объекта *Binding*

На рис. 32.17 показано, что происходит, когда объект Binding добавляется к Control. Привязка устанавливает связь между элементом управления и источником данных, которая поддерживается внутри BindingContext формы (или самого элемента управления). Изменения внутри источника данных отображаются в элементе управления, как если бы они были выполнены в нем самом.

Binding

Этот класс связывает свойство элемента управления с членом источника данных. Когда член изменяется, свойство элемента управления обновляется, чтобы отобразить это изменение. Также верно противоположное — если обновляется текст в текстовом поле, это изменение отображается в источнике данных.

Привязка может быть выполнена из любого столбца на любое свойство элемента управления. Например, вы можете привязать не только текст в текстовом поле, но также и его цвет. Можно привязать свойства элемента управления к совершенно различным источникам данных; например, цвет ячейки может быть задан в таблице цветов, тогда как данные — в другой таблице.

CurrencyManager и PropertyManager

Когда создается объект Binding, также создаются соответствующие объекты CurrencyManager и PropertyManager, предоставляя возможность первоначальной привязки данных источника. Назначение этого класса — определять позицию текущей записи внутри источника данных, а также координировать все первоначальные привязки при изменении текущей записи. На рис. 32.18 показаны два поля из таблицы Products, а также способ перемещения между записями с помощью элемента управления TrackBar.

Ниже представлен основной код ScrollingDataBinding.

```
namespace ScrollingDataBinding
{
    partial class Form1: Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private DataSet CreateDataSet()
        {
            string customers = "SELECT * FROM Products";
            DataSet ds = new DataSet();
            using (SqlConnection con = new SqlConnection (
                ConfigurationSettings.
                    ConnectionStrings["northwind"].ConnectionString))
            {
                SqlDataAdapter da = new SqlDataAdapter(customers, con);
                da.Fill(ds, "Products");
            }
            return ds;
        }
    }
}
```



Рис. 32.18. Поля из таблицы Products и способ перемещения между записями

```

private void trackBar_Scroll(object sender, EventArgs e)
{
    this.BindingContext[ds, "Products"].Position = trackBar.Value;
}
private void retrieveButton_Click(object sender, EventArgs e)
{
    retrieveButton.Enabled = false;
    ds = CreateDataSet();
    textName.DataBindings.Add("Text", ds, "Products.ProductName");
    textQuan.DataBindings.Add("Text", ds, "Products.QuantityPerUnit");
    trackBar.Minimum = 0;
    trackBar.Maximum = this.BindingContext[ds, "Products"].Count - 1;
    textName.Enabled = true;
    textQuan.Enabled = true;
    trackBar.Enabled = true;
}
private DataSet ds;
}
}

```

Механизм прокрутки обеспечивается обработчиком событий `trackBar_Scroll`, который устанавливает позицию `BindingContext` на текущую позицию бегунка — изменение контекста привязки здесь обновляет данные, отображаемые на экране.

Данные привязаны к двум текстовым полям в обработчике событий `retrieveButton_Click` добавлением выражения привязки данных — здесь свойства `Text` элементов управления устанавливаются в значения полей источника данных. Можно привязать любое простое свойство элемента управления к элементу источника данных; например, вы можете привязать цвет текста, свойство доступности или любые другие.

Когда данные первоначально извлекаются, значение максимума бегунка прокрутки устанавливается равной количеству записей. Затем в методе прокрутки позиция `BindingContext` для `DataTable` устанавливается в текущую позицию бегунка. Это изменяет текущую запись `DataTable`, так что обновляются все элементы управления, привязанные к текущей записи (в данном примере — два текстовых поля).

Теперь, когда мы узнали, как привязать различные источники данных, такие как массивы, таблицы данных, представления данных, а также прочие разнообразные контейнеры данных, в следующем разделе рассмотрим, как расширения Visual Studio обеспечивают интеграцию доступа к данным в приложениях.

Visual Studio .NET и доступ к данным

В этом разделе мы опишем некоторые способы интеграции данных с графическим интерфейсом пользователя, которые обеспечивает Visual Studio, включая способы создания соединений, выбора некоторых данных, генерации `DataSet`, а также применение всех сгенерированных объектов для построения простого приложения.

Доступные инструментальные средства позволяют создать подключение к базе данных с помощью классов `OleDbConnection` или `SqlConnection`. Выбор класса зависит от типа используемой базы данных. После того, как соединение определено, можно создать `DataSet` и наполнить его прямо внутри Visual Studio .NET. При этом генерируется файл XSD для `DataSet`, а также код `.cs`. В результате получается `DataSet`, безопасный к типам.

Создание соединения

Для начала создадим новое Windows-приложение, затем — новое соединение с базой данных. Используя проводник серверов Server Explorer (рис. 32.19), можно управлять различными аспектами доступа к данным.

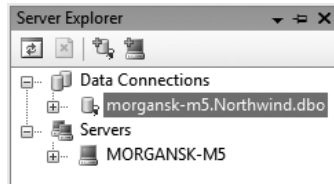


Рис. 32.19. Проводник Server Explorer

В нашем примере создадим соединение с базой данных Northwind. Для этого выберем пункт Add Connection (Добавить соединение) из контекстного меню, связанного с элементом Data Connections (Соединения с данными), для запуска мастера, который позволит выбрать поставщик базы данных. Выберем .NET Framework Provider for SQL Server (Поставщик .NET Framework для сервера SQL Server). На рис. 32.20 показано диалоговое окно Add Connection (Добавить соединение).

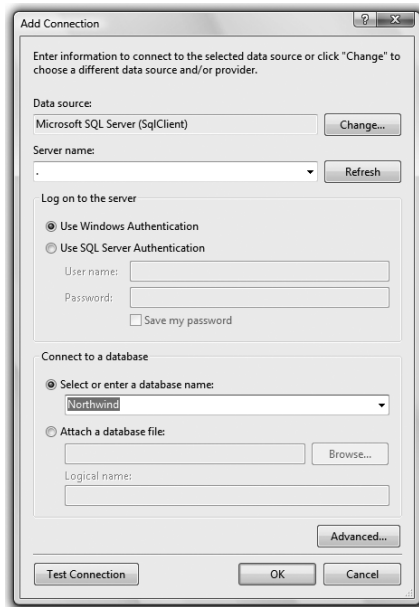


Рис. 32.20. Диалоговое окно Add Connection

В зависимости от инсталляции .NET Framework, база данных примеров может находиться на сервере SQL Server, MSDE (Microsoft SQL Server Data Engine) или на обоих. Чтобы подключиться к локальной базе MSDE, если она есть, следует ввести (local)\sqlservr в качестве имени сервера. Чтобы подключиться к экземпляру SQL Server, потребуется ввести (local) или . для выбора базы на текущей машине, либо имя нужного сервера в сети. Для доступа к данным может понадобиться указать имя пользователя и пароль.

Выберем базу Northwind из раскрывающегося списка баз данных и, убедившись, что все настроено правильно, щелкнем на кнопке Test Connection (Протестировать соединение). Если все правильно настроено, мы должны увидеть окно сообщения с соответствующим подтверждением.

Среда Visual Studio 2005 включала много новшеств, связанных с обращением к данным, и они доступны из нескольких мест пользовательского интерфейса. Меню Data (Данные) является одним из них — оно позволяет увидеть любые источники данных, уже добавленные к проекту, создавать новые, а также просматривать данные из лежащей в основе базы (или другого источника).

В следующем примере подключение к базе Northwind применяется для генерации пользовательского интерфейса выбора данных из таблицы Employee.

На первом шаге потребуется выбрать пункт Add New Data Source (Добавить новый источник данных) из меню Data (Данные), что запустит мастер, который проведет по всему процессу. Диалоговое окно, показанное на рис. 32.21, демонстрирует ту часть мастера Data Source Configuration Wizard (Мастер конфигурации источника данных), в которой можно выбрать подходящие таблицы для источников данных.

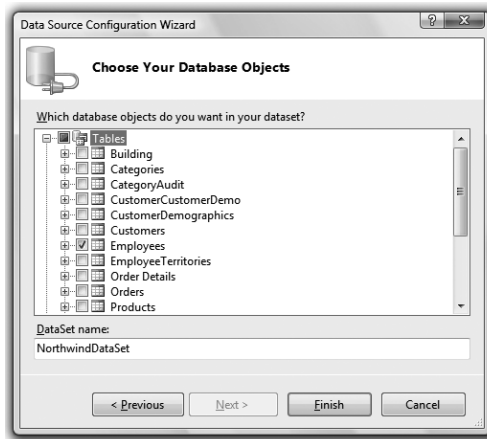


Рис. 32.21. Окно мастера Data Source Configuration Wizard

В процессе работы с мастером выбирается источник данных, который может быть базой данных, локальным файлом данных (таким как файл .mdb), Web-службой или объектом. Мастер предложит ввести дополнительную информацию в соответствии с выбранным источником данных. Для подключения к базе это будет имя соединения (которое потом сохраняется в конфигурационном файле приложения, представленном в следующем коде), затем можно выбрать таблицу, представление или хранимую процедуру, поставляющую данные. В конечном итоге это генерирует для приложения строго типизированный DataSet.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <connectionStrings>
    <add name="SimpleApp.Properties.Settings.NorthwindConnection"
      connectionString="Data Source=.;Integrated Security=True;Initial
      Catalog=Northwind"
      providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

Сюда включено имя соединения, сама строка соединения, а также имя поставщика, которое используется затем для генерации объекта соединения. Эту информацию при необходимости можно редактировать вручную. Чтобы отобразить пользовательский интерфейс данных о наемных сотрудниках, можно просто перетащить выбранные

данные из окна Data Sources (Источники данных) на проектируемую форму. При этом для нас будет сгенерирован один из двух стилей пользовательского интерфейса — табличный стиль (grid style), использующий элемент управления DataGridView, описанный ранее, либо представление Details (Подчиненный), показывающее данные по одной записи за раз. На рис. 32.22 показан внешний вид окна Data Sources при выборе представления.

Перетаскивание источника данных на форму генерирует множество объектов — как визуальных, так и невидимых. Невизуальные объекты создаются в области лотка формы и включают в себя DataConnector — строго типизированный DataSet, а также TableAdapter, который содержит SQL, применяемый для извлечения/обновления данных. Создаваемые визуальные объекты зависят от выбранного представления — DataGridView либо Details. Оба содержат элемент управления DataNavigator, который применяется для перемещения по записям. На рис. 32.23 показан пользовательский интерфейс, сгенерированный с применением элемента управления DataGridView, что отвечает целям Visual Studio 2005 настолько упростить доступ к данным, чтобы можно было генерировать функциональные формы вообще без написания каких-либо строк кода.

Когда создается источник данных, он добавляет множество файлов к решению. Чтобы увидеть их, необходимо щелкнуть на кнопке Show All Files (Показать все файлы) в Solution Explorer. Затем можно развернуть узел источников данных и увидеть добавленные файлы. Самый интересный из них — файл .Designer.cs, включающий исходный код на C#, используемый для наполнения набора данных.

Если заглянуть в файл .Designer.cs, можно обнаружить в нем несколько классов. Эти классы представляют строго типизированный набор данных, работающий способом, аналогичным стандартному классу DataAdapter. Внутри себя этот класс использует DataAdapter для наполнения DataSet.

Представленные классы — это строго типизированный набор данных, а также объект, реализующий определенный интерфейс.

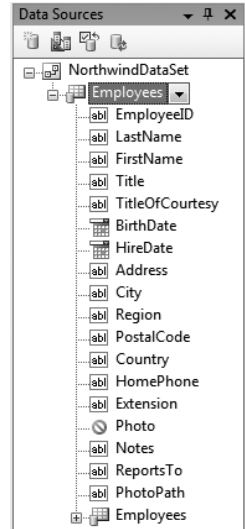


Рис. 32.22. Выбор представления в окне Data Sources

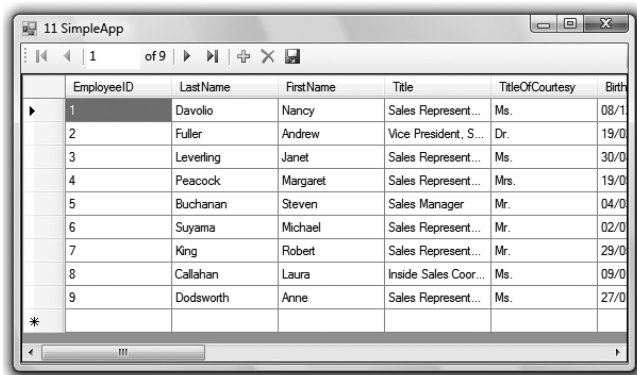


Рис. 32.23. Пользовательский интерфейс, сгенерированный с применением элемента управления DataGridView

Извлечение данных

Сгенерированный табличный адаптер содержит команды для SELECT, INSERT, UPDATE и DELETE. Излишне говорить, что они могут (и, возможно, должны) вызывать хранимые процедуры вместо прямого применения операторов SQL. Однако сгенерированный мастером код пока так и делает. Visual Studio .NET добавляет следующий код в файл .Designer:

```
private System.Data.SqlClient.SqlCommand m_DeleteCommand;  
private System.Data.SqlClient.SqlCommand m_InsertCommand;  
private System.Data.SqlClient.SqlCommand m_UpdateCommand;  
private System.Data.SqlClient.SqlDataAdapter m_adapter;
```

Объекты определены для всех команд SQL за исключением SELECT, а также SqlDataAdapter. Ниже в файле, в методе InitializeComponent(), мастер генерирует код создания каждой из этих команд, наряду с адаптером данных.

В предыдущих версиях Visual Studio .NET команды, сгенерированные для вставки и обновления, также включали конструкцию SELECT — это было сделано для повторной синхронизации данных с сервером на случай, если какие-то поля внутри базы данных были вычисляемыми (такие как identity и computed).

Сгенерированный мастером код работает, однако, он не оптимален. В промышленных системах весь сгенерированный код SQL должен быть заменен вызовами хранимых процедур. Если операторы INSERT и UPDATE не должны повторно синхронизировать данные, удаление излишних конструкций SQL повысит производительность приложения.

Обновление источника данных

До сих пор наши приложения только извлекали данные из базы. В этом разделе мы поговорим о том, как сохранять в базе измененные данные. Если следовать шагам, описанным в предыдущем разделе, мы получим приложение, которое содержит все необходимое, что должно быть в простейшем приложении. Для включения кнопки Save (Сохранить) потребуется внести лишь одно изменение в сгенерированную панель инструментов и написать обработчик событий, который обновит базу данных.

В среде IDE выберем кнопку Save (Сохранить) в навигационном элементе управления и изменим ее свойство Enabled на true. Затем, выполнив двойной щелчок на этой кнопке, сгенерируем обработчик события. Внутри него сохраним изменения данных, выполненные на экране, в базе данных:

```
private void dataNavigatorSaveItem_Click(object sender, EventArgs e)  
{  
    employeesTableAdapter.Update(employeesDataset.Employees);  
}
```

Поскольку Visual Studio выполнит за нас всю тяжелую работу, все, что мы должны сделать — использовать метод Update табличного адаптера; в этом примере используется перегрузка, принимающая в качестве параметра DataTable.

Другие общие требования

Общим требованием при отображении данных является показ контекстного меню для данной записи. Его можно реализовать разными способами. Пример, приведенный в этом разделе, сосредоточен на одном подходе, который может упростить необходимый код, особенно если контекст отображения — DataGridView, когда отображается DataSet с некоторыми отношениями. Проблема здесь состоит в том, что контекстное меню зависит от выбранной записи, а эта запись может быть частью любого источника DataTable в пределах DataSet.

Поскольку функциональность контекстного меню с высокой вероятностью по природе своей имеет общее назначение, здесь реализация использует базовый класс (`ContextDataRow`), поддерживающий код построения меню, и каждый класс строки данных (записи), поддерживающий всплывающее меню, наследуется от этого базового класса.

Когда пользователь щелкает правой кнопкой мыши на любой части строки в `DataGrid`, выполняется проверка, унаследована ли она от `ContextDataRow`, и если да, то может быть вызван метод `PopupMenu()`. Это можно было бы реализовать через интерфейс, однако в данном случае базовый класс обеспечивает более простое решение.

Настоящий пример демонстрирует, как генерировать классы `DataRow` и `DataTable`, которые можно использовать для обеспечения доступа, безопасного в отношении типов, к данным, почти таким же образом, как и в предыдущем примере с XSD. Однако на этот раз мы пишем код самостоятельно, чтобы показать, как в данном контексте использовать настраиваемые атрибуты и рефлексия.

На рис. 32.24 показана иерархия классов для этого примера.

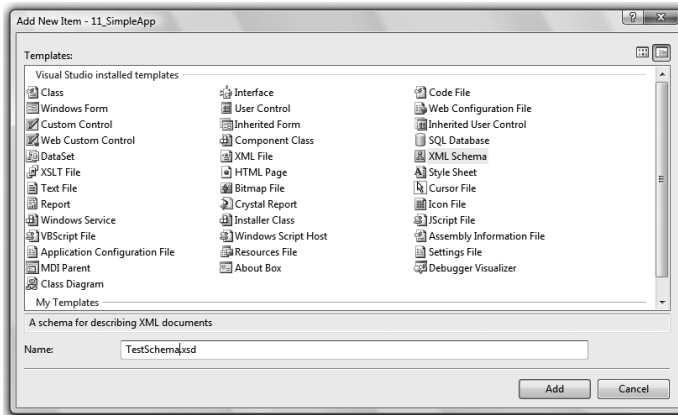


Рис. 32.24. Иерархия классов для примера с контекстным меню

Ниже приведен код этого примера.

```
using System;
using System.Windows.Forms;
using System.Data;
using System.Data.SqlClient;
using System.Reflection;
public class ContextDataRow : DataRow
{
    public ContextDataRow(DataRowBuilder builder) : base(builder)
    {
    }
    public void PopupMenu(System.Windows.Forms.Control parent, int x, int y)
    {
        // Использовать рефлексия, чтобы получить список команд
        // для всплывающего меню
        MemberInfo[] members = this.GetType().FindMembers (MemberTypes.Method,
            BindingFlags.Public | BindingFlags.Instance ,
            new System.Reflection.MemberFilter(Filter), null);
```

```
if (members.Length > 0)
{
    // Создать контекстное меню
    ContextMenu menu = new ContextMenu();
    // Теперь пройти по членам и сгенерировать всплывающее меню.
    // Обратите внимание на приведение к MethodInfo в цикле foreach
    foreach (MethodInfo meth in members)
    {
        // Получить заголовок операции из
        // ContextMenuAttribute
        ContextMenuAttribute[] ctx = (ContextMenuAttribute[])
            meth.GetCustomAttributes(typeof(ContextMenuAttribute), true);
        MenuCommand callback = new MenuCommand(this, meth);
        MenuItem item = new MenuItem(ctx[0].Caption,
            new EventHandler(callback.Execute));
        item.DefaultItem = ctx[0].Default;
        menu.MenuItems.Add(item);
    }
    System.Drawing.Point pt = new System.Drawing.Point(x, y);
    menu.Show(parent, pt);
}
}
private bool Filter(MemberInfo member, object criteria)
{
    bool bInclude = false;
    // Привести MemberInfo к MethodInfo
    MethodInfo meth = member as MethodInfo;
    if (meth != null)
    {
        if (meth.ReturnType == typeof(void))
        {
            ParameterInfo[] parms = meth.GetParameters();
            if (parms.Length == 0)
            {
                // Проверить, есть ли в методе ContextMenuAttribute у метода
                object[] atts = meth.GetCustomAttributes
                    (typeof(ContextMenuAttribute), true);
                bInclude = (atts.Length == 1);
            }
        }
    }
    return bInclude;
}
}
```

Класс `ContextDataRow` унаследован от `DataRow` и содержит две функции-члена: `PopupMenu()` и `Filter()`. Метод `PopupMenu()` использует рефлексии для поиска методов, соответствующих определенной сигнатуре, и отображает пользователю всплывающее меню этих опций. Метод `Filter()` применяется как делегат в `PopupMenu()` при перечислении методов. Он просто возвращает `true`, если функция-член соответствует соглашениям вызова:

```
MemberInfo[] members = this.GetType().FindMembers(MemberTypes.Method,
    BindingFlags.Public | BindingFlags.Instance,
    new System.Reflection.MemberFilter(Filter),
    null);
```

Это единственное выражение применяется для фильтрации всех методов текущего объекта и возврата только тех из них, которые отвечают следующим критериям:

- ❑ член должен быть методом;
- ❑ член должен быть общедоступным методом экземпляра (не статическим);
- ❑ член должен возвращать `void`;
- ❑ член должен принимать ноль параметров;
- ❑ член должен включать `ContextMenuAttribute`.

Последний из этих критериев ссылается на заказной атрибут, написанный специально для данного примера (мы говорили о нем после обсуждения метода `PopupMenu`).

```
ContextMenu menu = new ContextMenu();
foreach (MethodInfo meth in members)
{
    // ... Добавить элемент меню
}
System.Drawing.Point pt = new System.Drawing.Point(x, y);
menu.Show(parent, pt);
```

Экземпляр контекстного меню создан с элементами, добавленными для каждого из методов, отвечающих указанным критериям. Затем это меню отображается, как показано на рис. 32.25.

Главная сложность этого примера заключена в следующем фрагменте кода, повторенном по одному разу для каждой функции-члена, которая должна быть отображена во всплывающем меню:

```
System.Type ctxtype =
    typeof(ContextMenuAttribute);
ContextMenuAttribute[]
    ctx = (ContextMenuAttribute[])
        meth.GetCustomAttributes(ctxtype, true);
MenuCommand callback = new MenuCommand(this, meth);
MenuItem item = new MenuItem(ctx[0].Caption, new EventHandler(callback.Execute));
item.DefaultItem = ctx[0].Default;
menu.MenuItems.Add(item);
```

Каждый метод, который должен быть отображен в контекстном меню, снабжен атрибутом `ContextMenuAttribute`. Он определяет дружественное к пользователю имя команды меню, поскольку имена методов C# не могут включать пробелов, к тому же во всплывающем меню разумно использовать естественный язык вместо имен внутреннего кода. Атрибут извлекается из метода, и новый элемент меню создается и добавляется к коллекции команд всплывающего меню.

Пример кода также демонстрирует применение упрощенного класса `Command` (общий проектный шаблон). Класс `MenuCommand`, используемый в данном экземпляре, инициируется пользователем при выборе элемента контекстного меню и передает вызов приемнику метода — в данном случае, объекту и его методу, снабженному атрибутом. Это также позволяет изолировать код объекта-приемника от кода пользовательского интерфейса. Код объясняется в последующих разделах.

Таблицы и записи, созданные вручную

Ранее приведенный в этой главе пример XSD показывал код набора классов доступа к данным, созданный путем генерации средствами редактора Visual Studio .NET. Следующий класс показывает необходимые методы `DataTable`, которые предельно минимизированы (и написаны вручную).

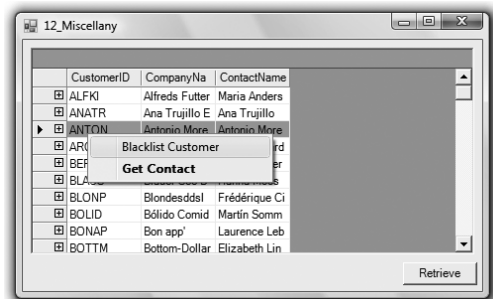


Рис. 32.25. Результирующее контекстное меню

```
public class CustomerTable : DataTable
{
    public CustomerTable() : base("Customers")
    {
        this.Columns.Add("CustomerID", typeof(string));
        this.Columns.Add("CompanyName", typeof(string));
        this.Columns.Add("ContactName", typeof(string));
    }
    protected override System.Type GetRowType()
    {
        return typeof(CustomerRow);
    }
    protected override DataRow NewRowFromBuilder(DataRowBuilder builder)
    {
        return (DataRow) new CustomerRow(builder);
    }
}
```

Первое требование — `DataTable` должен переопределить метод `GetRowType()`. Он используется внутри .NET при генерации новых записей таблицы. Этого достаточно для минимальной реализации. Соответствующий класс `CustomerRow` достаточно прост. Он реализует свойства для каждой из столбцов записи и затем реализует методы, которые в конечном итоге отображаются в контекстном меню:

```
public class CustomerRow : ContextDataRow
{
    public CustomerRow(DataRowBuilder builder) : base(builder)
    {
    }
    public string CustomerID
    {
        get { return (string)this["CustomerID"]; }
        set { this["CustomerID"] = value; }
    }
    // Прочие свойства опущены для ясности
    [ContextMenu("Blacklist Customer")]
    public void Blacklist()
    {
        // Делать что-то
    }
    [ContextMenu("Get Contact", Default=true)]
    public void GetContact()
    {
        // Делать что-то еще
    }
}
```

Класс просто наследуется от `ContextDataRow`, включая соответствующие методы `get/set` для свойств, чьи имена совпадают с именами полей, а затем добавляет методы, которые применяются средствами рефлексии при построении контекстного меню:

```
[ContextMenu("Blacklist Customer")]
public void Blacklist()
{
    // Делать что-то
}
```

Каждый отображаемый в контекстном меню метод имеет одну и ту же сигнатуру и включает заказной атрибут `ContextMenu`.

Использование атрибута

Идея написания атрибута `ContextMenuAttribute` заключается в том, чтобы можно было применять произвольное текстовое имя для данной команды меню. Следующий пример также добавляет флаг `Default`, используемый для пометки команды меню по умолчанию. Полный класс атрибута представлен ниже.

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple=false, Inherited=true)]
public class ContextMenuAttribute : System.Attribute
{
    public ContextMenuAttribute(string caption)
    {
        Caption = caption;
        Default = false;
    }
    public readonly string Caption;
}
```

Атрибут `AttributeUsage` класса помечает `ContextMenuAttribute` как единственный используемый атрибут метода, а также указывает, что для каждого данного метода может существовать только один экземпляр этого атрибута. Конструкция `Inherited=true` говорит о том, что атрибут может быть помещен в метод суперкласса и наследоваться его подклассами.

К этому атрибуту можно добавить ряд других членов, включая следующие:

- ☐ горячая клавиша выбора меню;
- ☐ отображаемый графический образ;
- ☐ некоторый текст, который может быть показан в строке состояния при наведении указателя мыши на команду меню;
- ☐ идентификатор контекст подсказки.

Диспетчеризация методов

Когда меню отображается в .NET, каждая его команда привязана к обрабатывающему коду с помощью делегата. Существуют два варианта реализации механизма подключения кода к командам меню.

- ☐ Реализовать метод с такой же сигнатурой, как у `System.EventHandler`. Это определено в следующем фрагменте:

```
public delegate void EventHandler(object sender, EventArgs e);
```

- ☐ Определить прокси-класс, реализующий предыдущий делегат, и перенаправить вызов классу-приемнику. Это известно, как шаблон `Command`, и именно такой вариант был использован в рассмотренном примере.

Шаблон `Command` (Команда) разделяет отправителя и приемника вызова посредством простого класса-посредника. Он может показаться избыточным для данного примера, но это упрощает методы каждого `DataRow` (поскольку им не нужно передавать делегату параметры) и обеспечивает возможность расширения:

```
public class MenuCommand
{
    public MenuCommand(object receiver, MethodInfo method)
    {
        Receiver = receiver;
        Method = method;
    }
}
```

```

public void Execute(object sender, EventArgs e)
{
    Method.Invoke(Receiver, new object[] { } );
}
public readonly object Receiver;
public readonly MethodInfo Method;
}

```

Данный класс просто предоставляет делегат EventHandler (метод Execute), который вызывает требуемый метод объекта-приемника. Этот пример обрабатывает два разных типа записей: записи из таблицы Customers и записи из таблицы Orders. Естественно, обработка опций для каждого из этих типов отличается. На рис. 32.25 показаны действия, доступные для записи Customer, а на рис. 32.26 — для записи Order.

Получение выбранной записи

Последний элемент мозаики этого примера, который мы рассмотрим — как определить, какая запись из DataSet выбрана пользователем. Вы можете подумать, что это должно быть свойством DataGridView. Однако этот элемент управления в данном контексте недоступен. Информация о попадании, полученная в обработчике события MouseUp(), также может рассматриваться как средство обнаружения выбранной записи, но это помогает только при условии, что отображаются данные из единственной DataTable. Вспомним, как заполнялась экранная таблица:

```
dataGrid.SetDataBinding(ds, "Customers");
```

Этот метод добавляет CurrencyManager к BindingContext, который представляет текущую DataTable в DataSet. Теперь DataGridView имеет два свойства — DataSource и DataMember, которые устанавливаются при вызове SetDataBinding(). DataSource в экземпляре ссылается на DataSet, а свойство DataMember ссылается на Customers.

Имея источник данных, член данных и контекст привязки формы, текущая запись может быть найдена с помощью следующего кода:

```

protected void dataGrid_MouseUp(object sender, MouseEventArgs e)
{
    // Выполнить проверку попадания
    if(e.Button == MouseButtons.Right)
    {
        // Найти строку, на которой щелкнул пользователь
        DataGridView.HitTestInfo hti = dataGrid.HitTest(e.X, e.Y);
        // Проверить попадание в ячейку
        if(hti.Type == DataGridView.HitTestType.Cell)
        {
            // Найти DataRow, который соответствует ячейке,
            // на которой щелкнул пользователь

```

После вызова dataGrid.HitTest() для вычисления места, на котором щелкнул пользователь, извлекается экземпляр BindingManagerBase:

```

BindingManagerBase bmb = this.BindingContext[dataGrid.DataSource,
    dataGrid.DataMember];

```

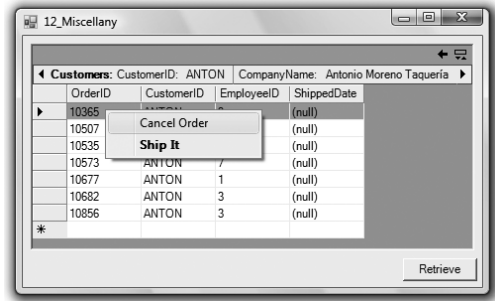


Рис. 32.26. Действия, доступные для записи Order

Этот код использует `DataSource` и `DataMember`, принадлежащие `DataGrid`, для именования возвращаемого объекта. Все, что остается — найти запись, на которой щелкнул пользователь, и отобразить контекстное меню. Когда выполняется щелчок правой кнопкой мыши на записи, то обычно индикатор текущей записи не перемещается, однако это не хорошо. Указатель записи должен быть перемещен перед тем, как будет отображено всплывающее меню. Объект `HitTestInfo` включает номер записи, поэтому текущую позицию объекта `BindingManagerBase` можно изменить следующим образом:

```
bmb.Position = hti.Row;
```

Это изменяет индикатор ячейки и в то же время означает, что когда выполняется вызов в класс для получения `Row`, то возвращается текущая запись, а не та, что была выбрана перед этим:

```
DataRowView drv = bmb.Current as DataRowView;
if (drv != null)
{
    ContextDataRow ctx = drv.Row as ContextDataRow;
    if (ctx != null) ctx.PopupMenu(dataGrid, e.X, e.Y);
}
}
```

Поскольку `DataGrid` отображает элементы `DataSet`, объект `Current` внутри коллекции `BindingManagerBase` является `DataRowView`, что в предыдущем коде проверяется с помощью явного приведения. Если оно проходит успешно, то действительная строка, помещенная в оболочку `DataRowView`, может быть извлечена посредством еще одного приведения для проверки того, что это действительно `ContextDataRow`, после чего появится меню.

Обратите внимание, что в примере были созданы две таблицы — `Customers` и `Orders`, и между ними было определено отношение, так что когда пользователь щелкает на `CustomerOrders`, он видит отфильтрованный список заказов.

Когда пользователь выполняет щелчок, `DataGrid` изменяет `DataMember` с `Customers` на `Customers.CustomerOrders`; это обеспечивает то, что для показа извлекаемых данных используется правильный объект, определенный индексатором `BindingContext`.

Резюме

В настоящей главе были представлены некоторые методы отображения данных в классах `.NET System.Windows.Forms`, включая больше количество классов, которые можно исследовать самостоятельно. Также здесь использовались элементы управления `DataGridView` и `DataGrid` для отображения данных из множества разнообразных источников — таких как `Array`, `DataTable` или `DataSet`.

Поскольку не всегда нужно отображать данные в табличном виде, здесь также было рассмотрено, как связать столбец данных с отдельным элементом управления в пользовательском интерфейсе. Эти средства привязки `.NET` обеспечивают возможность легкой поддержки пользовательского интерфейса подобного рода, поскольку обычно она заключается всего лишь в привязке элемента управления к столбцу, а всю сложную работу выполняет `.NET`.

В следующей главе речь пойдет о выводе графики посредством `GDI+`.