

6

Typed DataSets

IN CHAPTER 5, I extolled the virtues of setting up your DataSets like in-memory databases. Unfortunately, I asked you to write quite a bit of code to do all the work. I was only teasing you. This chapter will show you how to use Typed DataSets to make that job a lot easier, while at the same time creating type-safety at compilation time.

6.1 What Are Typed DataSets?

Typed DataSets are a different animal than most of what we have discussed so far. They are not a set of classes in the framework, but instead, they are a set of generated classes that inherit directly from the DataSet family of classes. Figure 6.1 contains the class diagram from Chapter 5, which shows how the elements of the DataSet are related.

In contrast, the Typed DataSet derives from these classes. The class diagram looks like that shown in Figure 6.2.

But why are they called Typed DataSets? In Chapter 5 we saw that we could create DataColumn for our DataTables to specify what type of data could be stored in each column. This enforces runtime type-safety, but on most occasions we would like to know that our DataSets are type-safe when we write the code. Typed DataSets generate classes that expose each object in a DataSet in a type-safe manner. With a DataSet, our code would look like Listing 6.1.

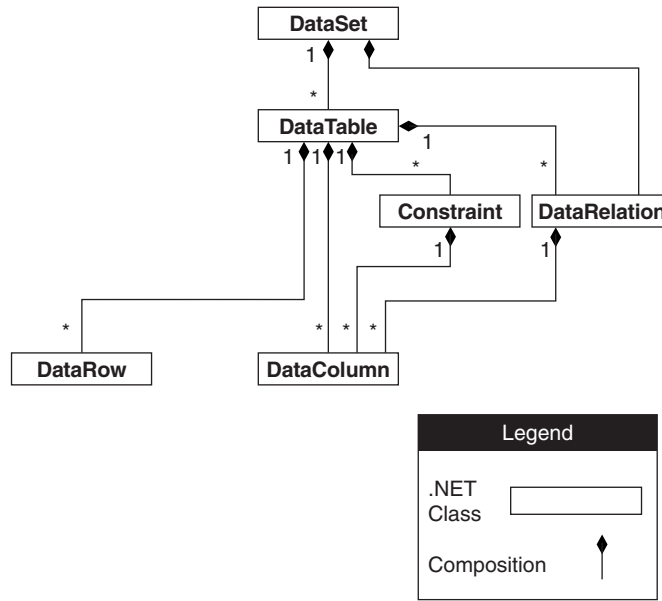


Figure 6.1: The structure of a DataSet

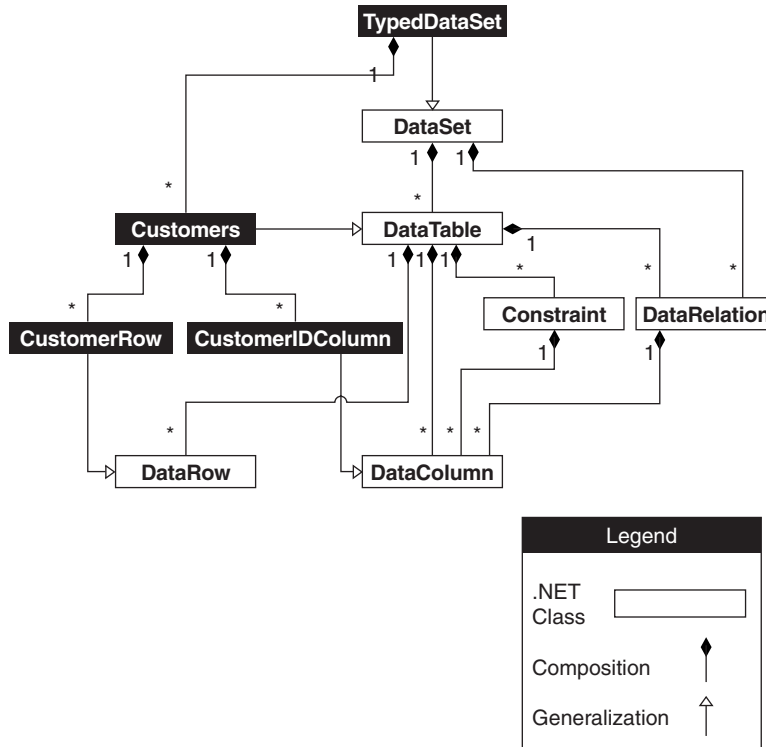


Figure 6.2: The structure of a Typed DataSet

Listing 6.1: *Using a DataSet*

```
...

// Create a DataAdapter for each of the tables we're filling
SqlDataAdapter daCustomers =
    new SqlDataAdapter("SELECT * FROM CUSTOMER;", conn);

// Create the Invoice DataAdapter
SqlDataAdapter daInvoices =
    new SqlDataAdapter("SELECT * FROM INVOICE", conn);

// Create your blank DataSet
DataSet dataSet = new DataSet();

// Fill the DataSet with each DataAdapter
daCustomers.Fill(dataSet, "Customers");
daInvoices.Fill(dataSet, "Invoices");

// Show the customer name
Console.WriteLine(dataSet.Tables["Customers"].
    Rows[0]["FirstName"].ToString());
Console.WriteLine(dataSet.Tables["Customers"].
    Rows[0]["LastName"].ToString());
Console.WriteLine(dataSet.Tables["Customers"].
    Rows[0]["HomePhone"].ToString());

// Change an invoice number with a string
// this shouldn't work because InvoiceNumber
// expects an integer
dataSet.Tables["Invoices"].
    Rows[0]["InvoiceNumber"] = "15234";
```

We need to use indexers with the DataSet to get each piece of the hierarchy until we finally get down to the row level. At any point you can misspell any name and get an error when this code is executed. In addition, the last line of the example shows us attempting to set the invoice number using a string. The DataSet knows that this column can only hold integers so we will get a runtime error enforcing that rule. In Listing 6.2, we do the same thing with a Typed DataSet.

Listing 6.2: Using a Typed DataSet

```

// Create a DataAdapter for each of the tables we're filling
SqlDataAdapter daCustomers =
    new SqlDataAdapter("SELECT * FROM CUSTOMER;", conn);

// Create the invoice DataAdapter
SqlDataAdapter daInvoices =
    new SqlDataAdapter("SELECT * FROM INVOICE", conn);

// Create your blank DataSet
CustomerTDS dataSet = new CustomerTDS();

// Fill the DataSet with each DataAdapter
daCustomers.Fill(dataSet, "Customers");
daInvoices.Fill(dataSet, "Invoices");

// Show the customer name
Console.WriteLine(dataSet.Customer[0].FirstName);
Console.WriteLine(dataSet.Customer[0].LastName);
Console.WriteLine(dataSet.Customer[0].HomePhone);
Console.WriteLine(DataSet.Customer[0].FullName);

// This will not compile because InvoiceNumber expects
// an integer
dataSet.Invoice[0].InvoiceNumber = "12345";

```

There are a few things to notice in this example. First, we create our Typed DataSet much like we created a DataSet in the first example—the difference is that the schema already exists in our Typed DataSet. Second, even though this is a Typed DataSet, the `CustomerTDS` class directly derives from the `DataSet` class. Therefore, when we call the `DataAdapters` to fill our `DataSet`, it will accept our Typed DataSet. In fact, the Typed DataSet is a `DataSet` . . . a specialized `DataSet`. Next, you should notice that the syntax to get at tables and fields is much more straightforward with Typed DataSets. Each `DataTable` is now referenced with a property of `CustomerTDS`. Likewise, each field is a property of a row. Not only is this syntax more straightforward, but you will get compiler errors if you misspell any of the elements. Lastly, when we try to set our invoice number with a string we also get a compiler error, because our generated class knows that invoice numbers are integers.

In addition to normal columns, you can set up expression columns in our Typed DataSets to make sure that expressions can be returned in a type-safe manner. For instance, in the above example, we can retrieve the `FullName` property from `Customer`. `FullName` is just an expression field that puts our customer's first and last names together in a convenient form. Because it is part of the Typed DataSet, this expression is returned as a string.

Lastly, as we will see in this chapter, using Typed DataSets as the basis for data object or business object layers is a powerful tool. By deriving directly from Typed DataSets we can eliminate much of the tedium of writing these layers, while at the same time achieving the type-safety we want.

6.2 Generating Typed DataSets

Now that you understand what Typed DataSets are all about, let's create our own. Typed DataSets can be generated in two ways: from within Visual Studio .NET or with the `XSD.exe` command-line tool. I will start with the Visual Studio .NET solution.

6.2.1 Using Visual Studio .NET to Create a Typed DataSet

This section will walk you through creating a Typed DataSet in Visual Studio .NET. To get started, please create a new Console C# project, as shown in Figure 6.3. Next, in the Solution Explorer add a new item, as shown in Figure 6.4.

From the Data folder in the Local Project Items, select a new DataSet and name it `ADONET.xsd`, as shown in Figure 6.5. If you add an XML Schema file instead (they both are .XSD files), the Typed DataSet will not generate. Make sure it is a DataSet. When you finish, the IDE will look something like what is shown in Figure 6.6.

You probably have noticed already that the Typed DataSet has an extension of `.xsd`. This is because the source of a Typed DataSet is an XML Schema Document. What the IDE asks you to do is to create a new .XSD file that represents the schema of the DataSet. This will include the schema information we discussed in Chapter 5. The XSD can include table and column names as well as keys, relationships, and constraints.

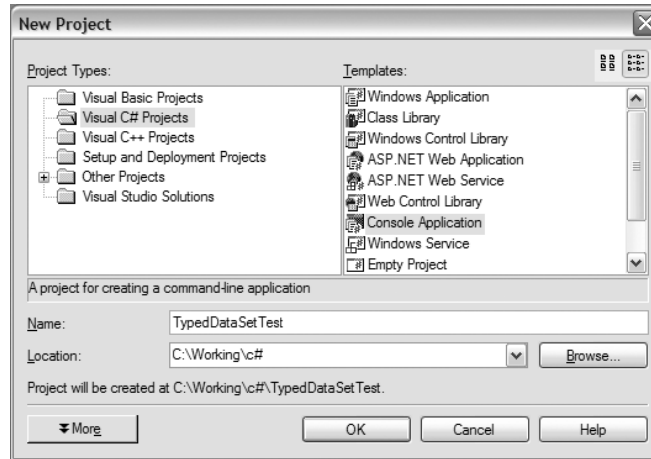


Figure 6.3: Creating a new C# project

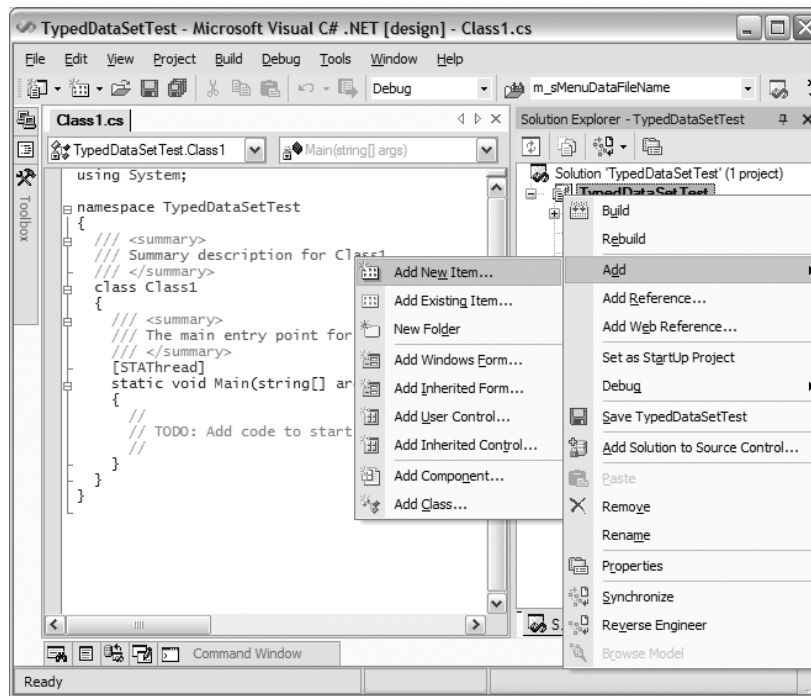


Figure 6.4: Adding a new item

6.2 GENERATING TYPED DATASETS 157

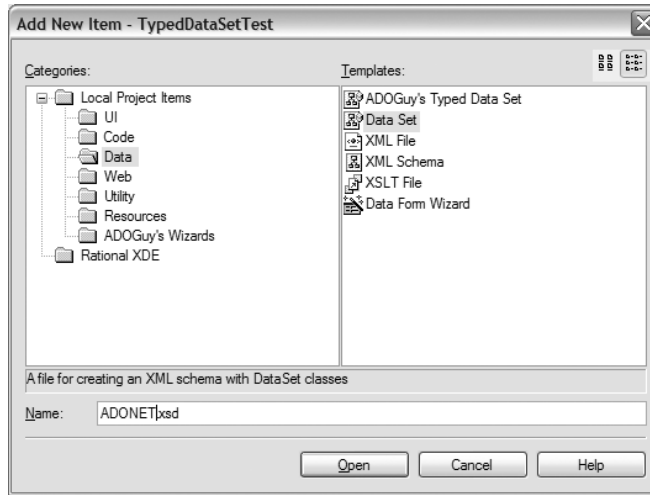


Figure 6.5: Adding a DataSet

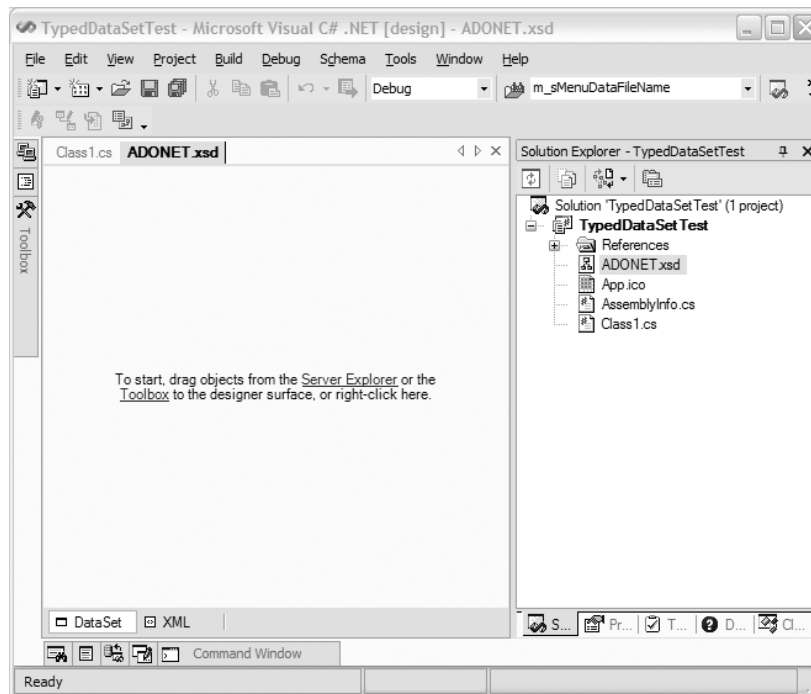


Figure 6.6: A blank canvas

Now that you have a new Typed DataSet added to your project, you can use either the Server Explorer to add tables to the DataSet or the Toolbox to add elements to the DataSet. Using the Server Explorer to add tables is shown in Figure 6.7.

After you have the Server Explorer open to an existing database, you can simply drag and drop the tables you want onto the .XSD file, to produce something that looks like Figure 6.8.

At this point, the new Typed DataSet will include two tables, but the tables remain unrelated. Unfortunately Visual Studio .NET does not try to discern all the schema information that may be contained in the database. In order to set the schema information, you will need to add it to the XSD.

We will start by adding a relationship between the `Customer` and `Invoice` tables. To do this, drag a new `Relation` from the Toolbox onto the `Customer` table, as shown in Figure 6.9. Once you do that, the `Edit Relation` dialog box appears, as shown in Figure 6.10.

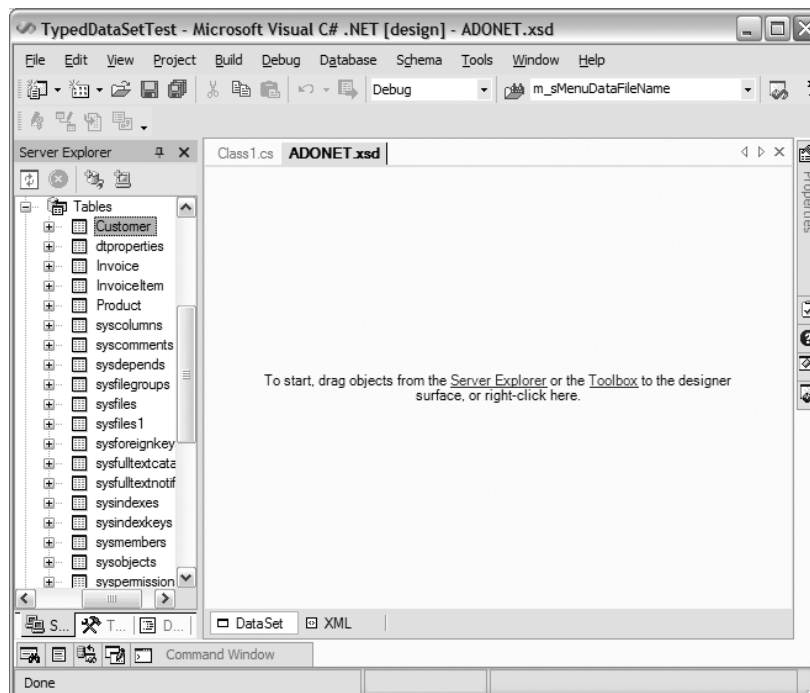


Figure 6.7: *The Server Explorer*

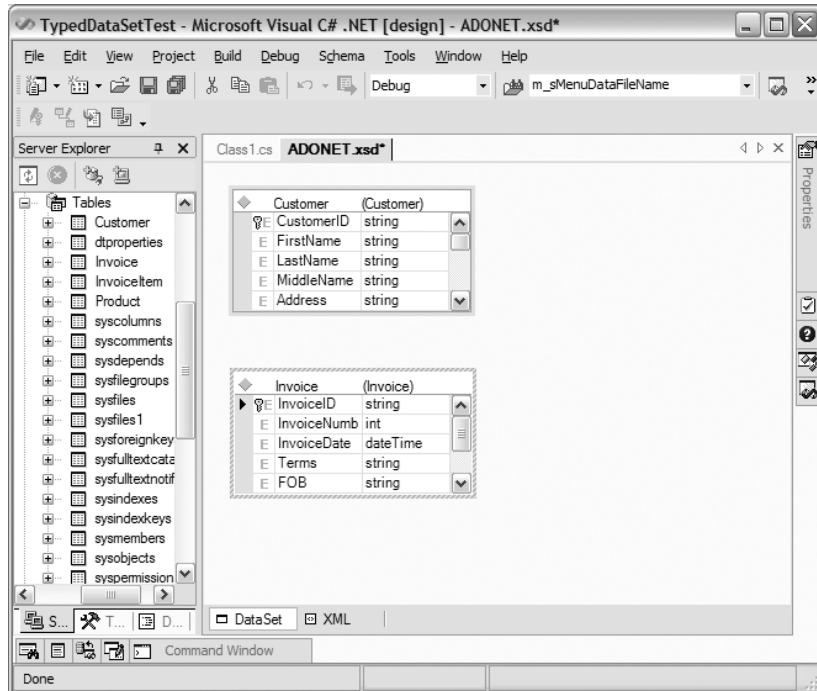


Figure 6.8: Tables in the .XSD file

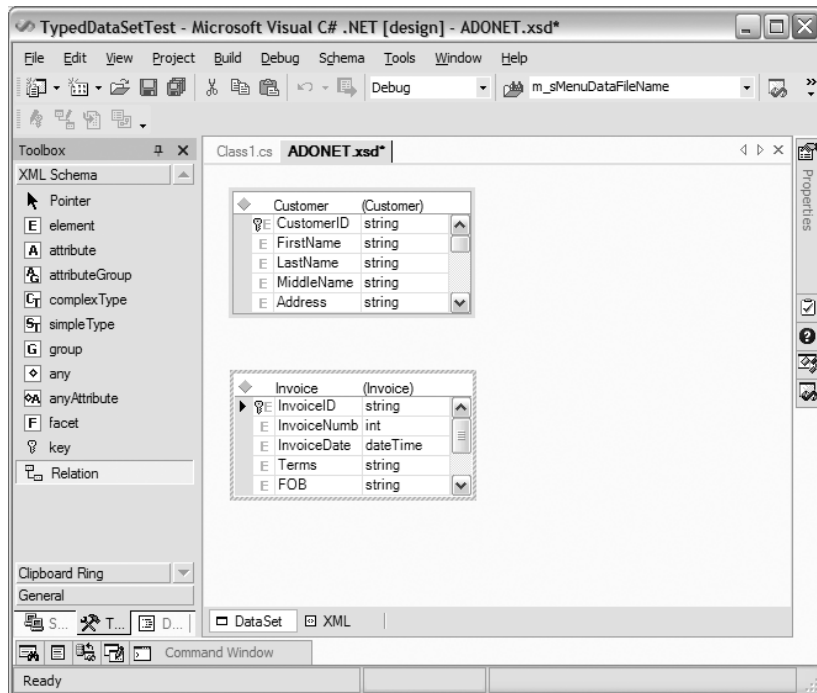


Figure 6.9: Adding a relationship

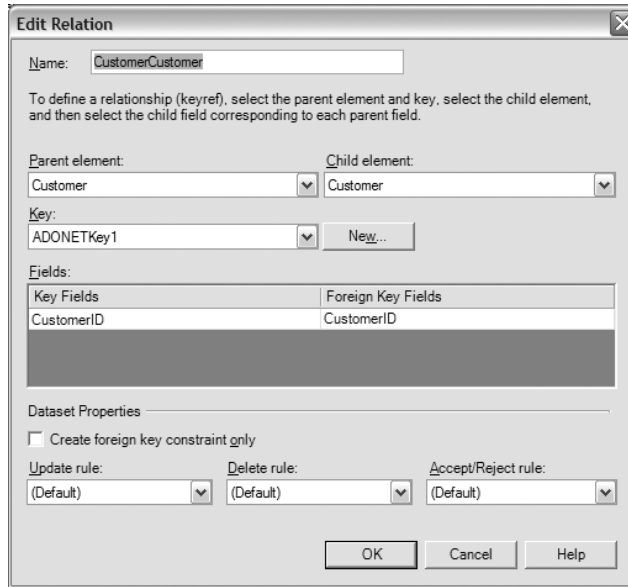


Figure 6.10: *The Edit Relation dialog box*

The Edit Relation dialog box is where you will provide the specifics of the relationship. You will need to change the child element to the child table (in this case *Invoice*). Usually the dialog box will correctly select the foreign key field(s). If necessary, you can change the way that the Update rule, Delete rule, and Accept/Reject rule settings affect cascading changes to the DataSet. After you make these changes you should see something similar to what is shown in Figure 6.11.

After you click OK, you will see the Relation, as shown in Figure 6.12.

Now that we have our tables and the relationship set up, let's add an expression column to our *Customer* table. To add an expression column, go to the bottom of the *Customer* table and add a new row. Name it *Full-Name* and look at the properties for the column. Set the expression of the column to:

```
LastName + ', ' + FirstName
```

Once you have done this, it should look like that shown in Figure 6.13.

Lastly, we need to add a unique constraint on *Home Phone* to make sure all of our customers' home phone numbers are unique. In the Typed

6.2 GENERATING TYPED DATASETS 161

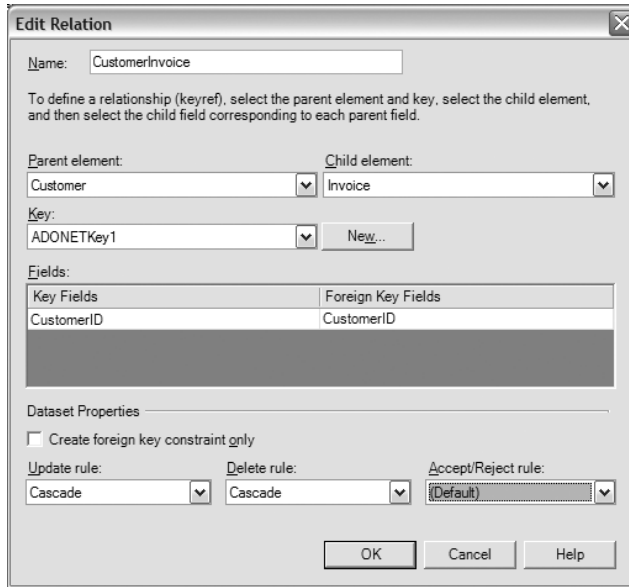


Figure 6.11: The completed Edit Relation dialog box

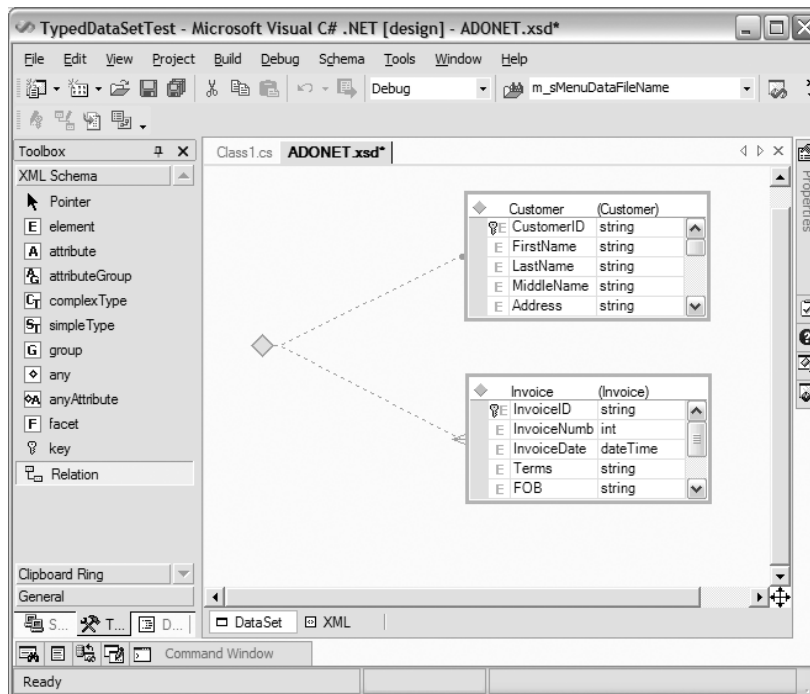


Figure 6.12: Our tables with a new relationship

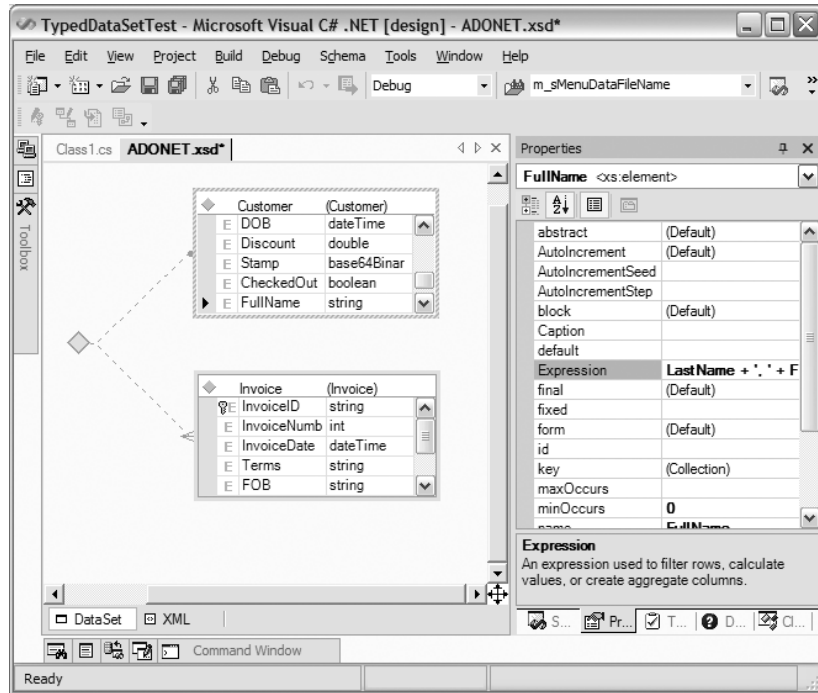


Figure 6.13: Adding an expression column

DataSet vernacular, you will need to add a key to that column. Select the column, right-click on it, and select Add Key, as shown in Figure 6.14. This will launch the Edit Key dialog box, shown in Figure 6.15.

If you do nothing but name the key and click OK, you’ve created a unique constraint. There may be cases in which you want to support having a multi-column unique key. To do this you would add fields to the key in the dialog box, as shown in Figure 6.16.

This unique constraint will force the DataSet to make sure that all Home-Phone and BusinessPhone combinations are unique. Congratulations, you now have your Typed DataSet defined.

Before we can see the code that Visual Studio .NET will generate, we need to build the project. After you build the project, your new Typed DataSet will be available for use.

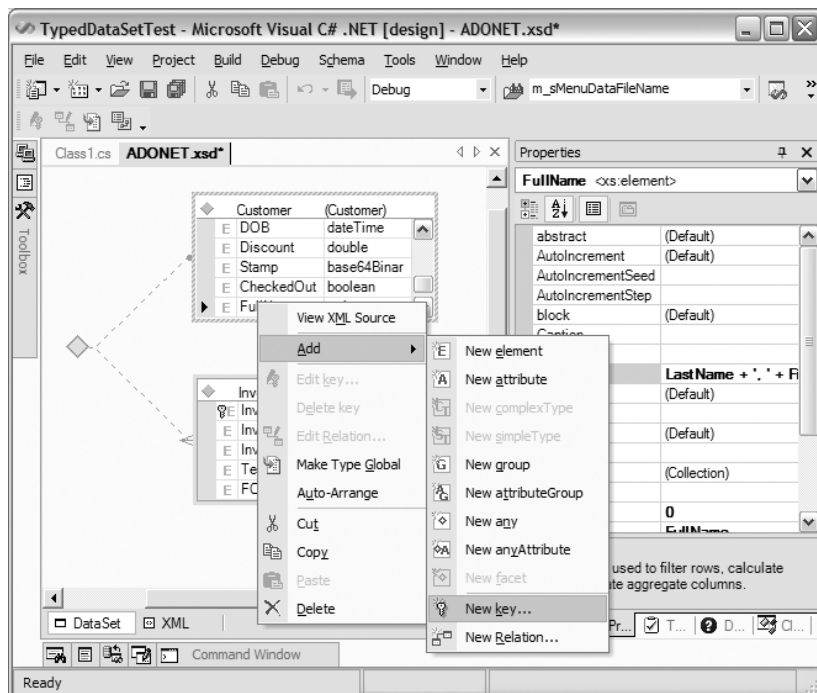


Figure 6.14: Selecting Add Key

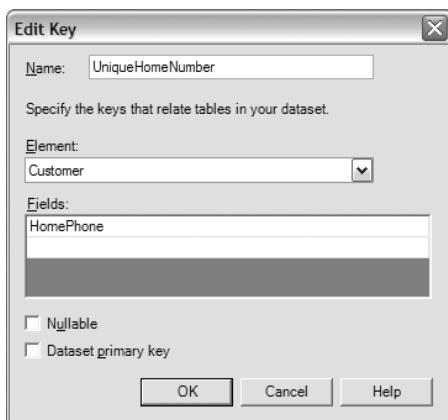


Figure 6.15: Naming the key

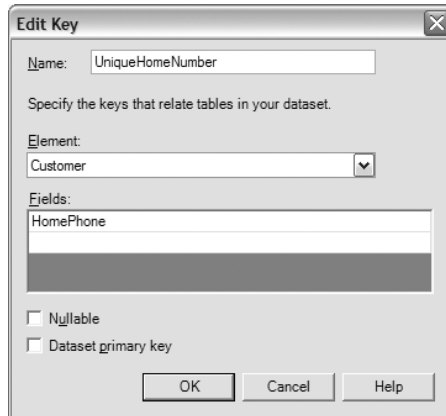


Figure 6.16: Supporting a multi-column unique key

6.2.2 Using XSD.exe to Create a Typed DataSet

Much like creating the class from Visual Studio .NET, Typed DataSets can also be created directly from an XML Schema Document (XSD) using the command-line XSD.exe tool that is included in the .NET Framework SDK. If you are not using Visual Studio .NET or just like your tools to be command-line tools, XSD.exe is for you. If you are using Visual Studio .NET, its built-in support for Typed DataSets is much easier than the tool and produces the same classes.

This tool is used to generate code from XML Schema Documents—both classes derived from the .NET XML classes and those derived from the DataSet family of classes. Before we can start, we will need an XSD document. We can use the DataSet to create one of these for us, as in Listing 6.3.

Listing 6.3: *Creating an XSD from a DataSet*

```
...

// Create a DataAdapter for each of the tables we're filling
SqlDataAdapter daCustomers =
    new SqlDataAdapter("SELECT * FROM CUSTOMER", conn);

// Create your blank DataSet
DataSet dataSet = new DataSet();
```

6.2 GENERATING TYPED DATASETS 165

```

// Fill the DataSet with each DataAdapter
daCustomers.Fill(dataSet, "Customers");

// Grab our DataTable for simplicity
DataTable customersTable = dataSet.Tables["Customers"];

// Improve the schema
customersTable.Columns["CustomerID"].ReadOnly = true;
customersTable.Columns["CustomerID"].Unique = true;
customersTable.Columns["LastName"].MaxLength = 50;
customersTable.Columns["LastName"].AllowDBNull = false;
customersTable.Columns["FirstName"].MaxLength = 50;
customersTable.Columns["FirstName"].AllowDBNull = false;
customersTable.Columns["MiddleName"].MaxLength = 50;
customersTable.Columns["State"].DefaultValue = "MA";
customersTable.Columns["State"].MaxLength = 2;
customersTable.Columns["HomePhone"].Unique = true;

// Write out our XSD file to use to generate our
// typed DataSet
dataSet.WriteXmlSchema(@"c:\CustomerDS.xsd");

```

Once we run this code, we will have a CustomersDS.xsd file to use with XSD.exe. To generate the class, simply go to the command line and use the XSD.exe tool, as shown in Figure 6.17.

The most interesting options are:

- /d specifies to create a schema class that is of type DataSet. For Typed DataSet you will always specify this flag.
- /l specifies the language in which to generate the classes—CS is C#, VB is VB .NET and JS is Jscript .NET.
- /n specifies the namespace to create our class in. This flag is optional.

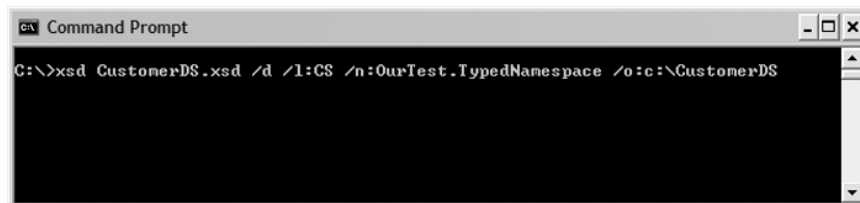


Figure 6.17: Running the XSD.exe tool from the command line

166 ■ PRAGMATIC ADO.NET

- `/o` specifies the output directory. If used, the output directory must already exist; the tool will not create it. If this is not specified, the classes will be created in the current directory.

Now you have your class file to include in your project.

6.2.3 Customizing the Generated Code with Annotations

ADO.NET allows you to control some aspects of the generated code by annotating the XSD that you use to generate the Typed DataSets. You can do this by annotating your XSD with several special attributes. All of these attributes are prefixed with the namespace of `codegen`. These attributes are:

- **typedName**: Specifies the name of an object.
- **typedPlural**: Specifies the name of the collection of objects.
- **typedParent**: Specifies the name of the parent relationship.
- **typedChildren**: Specifies the name of the child relationship.
- **nullValue**: Specifies how to handle a `DBNull` value for a particular row value. When you use this annotation, there are several possible values for the `nullValue` attribute:
 - **Replacement Value**: The value to return instead of a null (for example, `codegen:nullValue=""` will return an empty string instead of a null for a string field).
 - **_throw**: Throws an exception when the value is accessed and is null (for example, `codegen:nullValue=_throw`). If the user would use the `IsNull` properties as part of the Typed DataSet to determine whether the field is null before trying to retrieve it. This is the default behavior if not annotated.
 - **_null**: Returns a null reference from the field. If a value type is encountered an exception is thrown.
 - **_empty**: Returns a reference created with an empty constructor. For strings, it returns `String.Empty`. For value types, an exception is thrown.

These must be added directly to the `.XSD` file. If you are using Visual Studio .NET, you must go to the XML view of your XSD to add these annotations. These annotations are added to an XSD like so:

```

<?xml version="1.0" encoding="utf-8" ?>
<xs:schema id="AnnotatedTDS"
  targetNamespace="http://tempuri.org/AnnotatedTDS"
  elementFormDefault="qualified"
  attributeFormDefault="qualified"
  xmlns="http://tempuri.org/AnnotatedTDS"
  xmlns:mstns="http://tempuri.org/AnnotatedTDS"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
  xmlns:codegen="urn:schemas-microsoft-com:xml-msprop" >
<xs:element name="AnnotatedTDS" msdata:IsDataSet="true">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded">
      <xs:element name="Customer"
        codegen:typedName="OurCustomer"
        codegen:typedPlural="OurCustomers">
...
        <xs:element name="MiddleName" type="xs:string"
          minOccurs="0"
          codegen:nullValue="_empty" />
...
      </xs:element>
      <xs:element name="Invoice"
        codegen:typedName="TheirInvoice"
        codegen:typedPlural="TheirInvoices">
...
        <xs:element name="Terms" type="xs:string"
          minOccurs="0"
          codegen:nullValue="Net 30" />
...
      </xs:element>
    </xs:choice>
  </xs:complexType>
  <xs:keyref name="CustomerInvoice"
    refer="ADONETKey1"
    msdata:AcceptRejectRule="Cascade"
    msdata>DeleteRule="Cascade"
    msdata:UpdateRule="Cascade"
    codegen:typedChildren="Invoices"
    codegen:typedParent="Customer">
    <xs:selector xpath="//mstns:Invoice" />
    <xs:field xpath="mstns:CustomerID" />
  </xs:keyref>
...
</xs:element>
</xs:schema>

```

Let's walk through this XSD and see where we have put the annotations.

1. The first thing we needed to do was to add the namespace reference to the schema header. This simply tells the XSD that we have a namespace defined and to allow the attributes.
2. Next, we make some changes to the Customer Element. `typedName` changes the name of the Typed DataRow in the DataTable to `OurCustomer`. When you ask for an individual row from the DataTable, it will return you an instance of the `OurCustomer` class instead of the default of `Customer`. `typedPlural` changes the name of the property on the DataSet that returns the specific DataTable. When retrieving the Customer DataTable, you would call the `OurCustomers` property.
3. Next, we added the `codegen:nullValue="_empty"` to the `MiddleName` element. This annotated our XSD so that a `DBNull` in the middle name element should be treated as an empty string in our generated Typed DataSet.
4. Next, we changed the `Invoice` element just like we did the `Customer` element. We renamed the Typed DataRow to `TheirInvoice` and the property that returns it to `TheirInvoices`.
5. Last, we changed the names of each end of our relationship by annotating the `keydef` tag with `typedChildren` and `typedParent`. This changes how each side of the relationship is named. The `Customer` class will have a method called `Invoices` (instead of `GetInvoiceTable`) to navigate down the relationship and the `Invoice` class will have a method called `Customer` (instead of `GetCustomerTable`) to navigate up the relationship.

6.3 Using Typed DataSets

In most cases, you can use your Typed DataSet everywhere you would normally use a DataSet. Your Typed DataSet directly derives from DataSet so all the DataSet-related classes and methods will work. One of the distinct advantages of using a Typed DataSet is that elements of the DataSet are strongly typed and strongly named. The use of the new class is a little different than our old DataSet examples, as shown in Listing 6.4.

Listing 6.4: Using a Typed DataSet

```

...

// Create the Customer DataAdapter
SqlDataAdapter customerAdapter =
    new SqlDataAdapter("SELECT * FROM CUSTOMER", conn);

// Create the Invoice DataAdapter
SqlDataAdapter invoiceAdapter =
    new SqlDataAdapter("SELECT * FROM INVOICE", conn);

// Create the DataSet
CustomerTDS typedDS = new CustomerIDS();

// Use the DataAdapters to fill the DataSet
customerAdapter.Fill(typedDS, "Customer");
invoiceAdapter.Fill(typedDS, "Invoice");

// Show the address and # of invoices for each customer
foreach(CustomerTDS.CustomerRow custRow in typedDS.Customer)
{
    Console.WriteLine(custRow.FullName);
    Console.WriteLine(custRow.Address);
    Console.WriteLine("{0}, {1} {2}",
        custRow.City,
        custRow.State,
        custRow.Zip);
    Console.WriteLine("{0} (hm)",
        custRow.IsHomePhoneNull() ? "" :
        custRow.HomePhone);
    Console.WriteLine("{0} (bus)",
        custRow.IsBusinessPhoneNull() ? "" :
        custRow.BusinessPhone);
    Console.WriteLine("Invoices: " +
        custRow.GetChildRows("CustomerInvoice").Length);
    Console.WriteLine("");
}

```

When filling the Typed DataSet with data, you can deal with the new class just like a DataSet (because it derives directly from it). Once we fill it, we can use the typed accessors to get at rows and columns in a more direct way than with an untyped DataSet.

Because we allowed our `HomePhone` and `BusinessPhone` columns to be null, we need to check to see whether they are null before accessing them. If you do not do this, you will get a `TypedDataSetException` thrown unless you have used annotations to modify the generated code.

6.4 Simplification of Business Object Layers

I have spent more of my adult life writing business object layers than any other work I have done. The work always involved several tasks:

- Mapping our relational model to a hierarchical model.
- Reading and writing those objects to and from the database.
- Adding business logic to handle our business needs for the data.

Mapping the relational model to the hierarchical model and the database manipulation code were the most tedious aspects of this work. Luckily, by using `Typed DataSets` you can eliminate the need to write this code yourself. When you create a `Typed DataSet` with multiple tables and relationships, you already have your relational-to-object mapping. By viewing a single row in a single table as the top of an object graph, you can navigate the relationships to get a tree of items. This is similar to what was done when business object layers were written with database joins across multiple tables to get an object graph in a database result. In the `DataSet` and `Typed DataSet`, the difference is that if you navigate the relationships, you are navigating to only the related rows in the related table. In this way, the `Typed DataSet` relieves you from writing the relational-to-object mapping, because it is inherent in the way the `DataSet` works.

The next job of the database programmer in this situation is usually to write the database manipulation code. Because ADO.NET has abstracted the `DataSet` from the database manipulation code (primarily in the managed providers), you will not have to write much code to make the database manipulation work. You may still have to write the stored procedures to handle the CRUD (Create, Read, Update, Delete) operations to the database and tie those stored procedures to the `DataAdapter` (see Chapter 8 for more information on how this is done), but in the larger picture ADO.NET does a lot of the heavy lifting.

6.4 SIMPLIFICATION OF BUSINESS OBJECT LAYERS ■ 171

That leaves only the business logic or rules to write, which in my experience is usually the easiest of the code to write. What is this business logic exactly? In some systems, business logic is as simple as data validation, whereas in other systems it is as complicated as integrating various systems to keep them in sync. Business logic is really any logic that needs to be added to the raw data in the database. So in this model where we are using Typed DataSets to get around writing business logic layers, where do we put our business logic? I recommend deriving from your generated Typed DataSets to put this logic in place.

6.4.1 Hooking Up Your Business Logic

The code generated from the .XSD file is still just a class, so you can inherit from it very simply—but how do you hook up your business logic? There are two approaches, and both are appropriate in the right circumstances. Both approaches begin by inheriting directly from the generated Typed DataSet.

6.4.1.1 *Event-Driven DataSet*

In the case where you do not have much business logic, you can decide to derive from the Typed DataSet and register for event notification to do your business logic. Within the DataSet, you register for notification from many different events, but the ones that make the most sense for most business logic are the RowChanging and RowChanged events. Listing 6.5 is an example of that solution.

Listing 6.5: Capturing Events from a Typed DataSet

```
public class CustomersObject : CustomerTDS
{
    public CustomersObject() : base()
    {
        Register();
    }

    protected CustomersObject(SerializationInfo info,
                               StreamingContext context) :
        base(info, context)
    {
        Register();
    }
}
```

172 ■ PRAGMATIC ADO.NET

```

private void Register()
{
    Invoice.InvoiceRowChanging +=
        new InvoiceRowChangeEventHandler(InvoiceChanging);
}

private void InvoiceChanging(object source,
                             InvoiceRowChangeEvent args)
{
    if (args.Action == DataRowAction.Add ||
        args.Action == DataRowAction.Change)
    {
        if (args.Row.InvoiceDate > DateTime.Today)
        {
            throw new Exception("Cannot Create Invoices" +
                                "in the Future");
        }
    }
}
}

```

We create a new class that derives from our Typed DataSet (CustomerTDS) and create two constructors. The first of these is for normal construction, and the other is for XML deserialization. The constructor that is used for deserialization must be implemented or there will be no support for XML serialization. Other than calling the base class's constructors, the only other thing we do here is call our new Register method, which is used to register for the events we are interested in. In this case, we have registered for the InvoiceRowChangingEvent. This is fired before the change to the row actually takes place. In our handler method (InvoiceChanging), we check to see whether the changed row was added or changed. If either of these actions occurred, we make sure the invoice date does not occur in the future. If it does, we throw an exception. We would use our new Typed DataSet as in Listing 6.6.

Listing 6.6: Testing Our Event-driven Typed DataSet Business Logic

```

...

// Create a DataAdapter for each of the tables we're filling
SqlDataAdapter daCustomers =
    new SqlDataAdapter("SELECT * FROM CUSTOMER;", conn);

```

6.4 SIMPLIFICATION OF BUSINESS OBJECT LAYERS 173

```

// Create the invoice DataAdapter
SqlDataAdapter daInvoices =
    new SqlDataAdapter("SELECT * FROM INVOICE", conn);

// Create an instance of our inherited Typed DataSet
CustomersObject dataset = new CustomersObject();

// Use the DataAdapters to fill the DataSet
daCustomers.Fill(dataset, "Customer");
daInvoices.Fill(dataset, "Invoice");

// This will throw an exception because we're creating
// an invoice date in the future
CustomerTDS.InvoiceRow invoice;
Invoice = dataset.Invoice.AddInvoiceRow(Guid.NewGuid(),
    DateTime.Now + new TimeSpan(4,0,0,0),
    "",
    "",
    "",
    dataset.Customer[0]);

```

The use of our derived Typed DataSet is identical to how the original Typed DataSet is used. But because we have registered for events, we will be notified during certain types of operations to implement our business logic. In this case, when we add a new invoice, we are creating it with an invoice date four days in the future so the event will throw an exception to let the user know he did something wrong.

6.4.1.2 *Deriving from Typed DataSets*

The other approach is not only to derive from the Typed DataSet, but also to derive from the Typed DataTable and Typed DataRow classes. This allows us to override any behavior to implement our business logic. We do not have to derive from each and every DataTable or DataRow, just the ones that need specific business logic. If we are going to need to derive from the DataRow, we will need to derive from its parent DataTable.

For this example, we want to put some logic into the Invoice table to check for credit before we allow a new invoice to be added to the table. Ultimately, we want our logic to look something like Listing 6.7.

Listing 6.7: *AddInvoiceRow Method*

```

public void AddInvoiceRow(InheritedInvoiceRow row)
{
    if (DoesCustomerHaveCredit())
    {
        base.AddInvoiceRow(row);
    }
    else
    {
        throw new Exception(
            "Customer Invoice cannot be created, " +
            "no credit available");
    }
}

```

To get to the point where we can make this change, we will have to start by inheriting from the Typed DataTable, as shown in Listing 6.8.

Listing 6.8: *Inheriting from a Typed DataSet*

```

public class InheritedTDS : FixedCustomerTDS
{
    ...

    public class InheritedInvoiceDataTable : InvoiceDataTable
    {
        internal InheritedInvoiceDataTable() : base()
        {
        }

        internal InheritedInvoiceDataTable(DataTable table) :
            base(table)
        {
        }

        ...

    }

    ...

}

```

6.4 SIMPLIFICATION OF BUSINESS OBJECT LAYERS ■ 175

Inheriting from the Typed DataTable requires that we support two constructors again. This time, the second constructor takes a DataTable. This second constructor is used for XML serialization as well. Creating the InheritedInvoiceDataTable was easy; the hard part is getting the Typed DataSet to use this class for its InvoiceDataTable. The generated code makes it hard on us because it is creating the entire schema (including our DataTable) during the base class's construction. This means that in order to replace the old DataTable with our inherited class we need to re-create much of the schema that is in the constructor. To get around this we could edit the generated code in a few small ways. First we could add a new virtual method that is called during construction to build the DataTable we want to replace, as shown in Listing 6.9.

Listing 6.9: Adding a Create Table Method

```
// This is the generated class
public class FixedCustomerTDS : DataSet
{
    ...

    protected virtual InvoiceDataTable
        CreateInvoiceDataTable(DataTable table)
    {
        if (table == null) return new InvoiceDataTable();
        else return new InvoiceDataTable(table);
    }

    ...
}
```

This method will return a Typed DataTable object. This is important because when we inherit from the Typed DataSet, we will need to override this method and return the same Typed DataTable. It ensures that our inherited DataTable actually does inherit from their Typed DataTable. The generated code will have code that depends on the DataTable being typed and by writing the method this way, we will guarantee not to break the generated code.

Next, we need to override this method when we inherit from the Typed DataSet. Because the method is virtual (or overridable in VB .NET), the

Typed DataSet will call our version of this method when it constructs the DataTables. To hook this up to our Typed DataSet, we will need to inherit from the Typed DataSet and override the creation method, as shown in Listing 6.10.

Listing 6.10: *Calling the Create Table Method*

```

public class InheritedTDS : FixedCustomerTDS
{
    public InheritedTDS() : base()
    {
    }

    protected InheritedTDS(SerializationInfo info,
                            StreamingContext context)
        : base(info, context)
    {
    }

    protected override InvoiceDataTable
        CreateInvoiceDataTable(DataTable table)
    {
        if (table == null)
        {
            return new InheritedInvoiceDataTable()
                as InvoiceDataTable;
        }
        else
        {
            return new InheritedInvoiceDataTable(table)
                as InvoiceDataTable;
        }
    }

    ...
}

```

This should look very much like the method in the base class, except that it creates our derived DataTable, but returns it as an instance of the base DataTable.

Next, we need to modify the generated code to replace all calls to construction of our DataTable to use this method (see Listing 6.11).

6.4 SIMPLIFICATION OF BUSINESS OBJECT LAYERS ■ 177

Listing 6.11: *Changing Default Behavior of the Typed DataSet*

```
// This is the generated class
public class FixedCustomerTDS : DataSet
{
    ...

    private void InitClass()
    {
        ...

        /* Originally
        this.tableInvoice = new InvoiceDataTable();
        */

        // New
        this.tableInvoice = CreateInvoiceDataTable(null);

        ...
    }
    ...

    protected FixedCustomerTDS(SerializationInfo info,
                               StreamingContext context)
    {
        ...

        /* Originally
        this.Tables.Add(
            new InvoiceDataTable(ds.Tables["Invoice"]));
        */

        // New
        this.Tables.Add(
            CreateInvoiceDataTable(ds.Tables["Invoice"]));

        ...
    }
    ...
}
```

We need to replace two different styles of construction. Each happens a couple times per Typed DataSet. The first style is `new NameDataTable()`. We want to replace this with `CreateNameDataTable()`. The second style is `new NameDataTable(SomeDataTable)`. We want to replace this with `CreateNameDataTable(SomeDataTable)`. In the above example, both styles are shown as they originally looked and as they looked after we changed them.

The last piece that will help make our derived Typed DataSet useful is to create a new property to hide the base class's `DataTable` property, as shown in Listing 6.12.

Listing 6.12: Hiding the DataTable Property

```
public class InheritedTDS : FixedCustomerTDS
{
    ...

    public new InheritedInvoiceDataTable Invoice
    {
        get
        {
            return (InheritedInvoiceDataTable)base.Invoice;
        }
    }

    ...
}
```

By creating a new `Invoice` property, we are not only hiding the base class's `Invoice` property, but also making sure all code that references the `Invoice` property is dealing with our derived version of the `DataTable`.

With all of this in place, we can put some business logic into our inherited `DataTable`, as shown in Listing 6.13.

Listing 6.13: Adding Our Business Logic to the Derived Class

```
public class InheritedTDS : FixedCustomerTDS
{
    ...
```

6.4 SIMPLIFICATION OF BUSINESS OBJECT LAYERS 179

```

public class InheritedInvoiceDataTable : InvoiceDataTable
{
    ...

    public void AddInvoiceRow(InheritedInvoiceRow row)
    {
        if (DoesCustomerHaveCredit())
        {
            base.AddInvoiceRow(row);
        }
        else
        {
            throw new Exception(
                "Customer Invoice cannot be created, " +
                "no credit available");
        }
    }

    ...
}

```

Now, when a user attempts to call the DataTable and add a new invoice for a customer, we will make sure that the customer can have a new invoice; otherwise, we can throw an exception. This should allow us to put business logic at the table level, but that may not be enough. We might want to control some different behavior at the row level. To ensure that none of our users create invoices that are accidentally dated in the future, we want the following business object added to the DataRow's InvoiceDate property (see Listing 6.14).

Listing 6.14: Protecting the Invoice Date

```

public new DateTime InvoiceDate
{
    get
    {
        return base.InvoiceDate;
    }
}

```

```

set
{
    if (value > DateTime.Today)
    {
        return new StrongTypingException("Invoice Date" +
            "cannot be in the" +
            "future", null);
    }
    else
    {
        base.InvoiceDate = value;
    }
}
}

```

To accomplish this, Listing 6.15 shows how we need to derive from the Typed DataRow as well.

Listing 6.15: Enabling Deriving from the DataRow

```

public class InheritedTDS : FixedCustomerTDS
{
    ...

    public class InheritedInvoiceRow : InvoiceRow
    {
        public InheritedInvoiceRow(DataRowBuilder builder) :
            base(builder)
        {
        }
    }

    ...
}

```

In the case of deriving from the Typed DataRow, the only constructor that is required is to support one that takes a DataRowBuilder. This con-

6.4 SIMPLIFICATION OF BUSINESS OBJECT LAYERS ■ 181

structor is used by the `DataTable.NewRowFromBuilder()` method. This method is called by the `DataTable` when a user asks for a new row for the `DataTable`. The base class calls the method to create rows that are type-safe. To make this work, Listing 6.16 shows how we need to override the `NewRowFromBuilder()` and `GetRowType()` methods in our `DataTable`.

Listing 6.16: Allowing Creation of New Derived DataRows

```
public class InheritedTDS : FixedCustomerTDS
{
    ...

    public class InheritedInvoiceDataTable : InvoiceDataTable
    {
        ...

        protected override DataRow
            NewRowFromBuilder(DataRowBuilder builder)
        {
            return new InheritedInvoiceRow(builder);
        }

        protected override System.Type GetRowType()
        {
            return typeof(InheritedInvoiceRow);
        }

        ...
    }

    ...
}
```

This ensures that all new rows created from this `DataTable` will be of our derived type (such as `InheritedInvoiceRow`). We need our `DataTable` to be handing out our new `DataRows` to users instead of the base class's implementation. To do this, we create a couple of new methods, as shown in Listing 6.17.

Listing 6.17: Overriding DataRow Creation

```
public class InheritedTDS : FixedCustomerTDS
{
    ...

    public class InheritedInvoiceDataTable : InvoiceDataTable
    {
        ...

        public new InheritedInvoiceRow this[int index]
        {
            get
            {
                return ((InheritedInvoiceRow)(this.Rows[index]));
            }
        }

        public new InheritedInvoiceRow
            AddInvoiceRow(Guid InvoiceID,
                        DateTime InvoiceDate,
                        string Terms,
                        string FOB,
                        string PO,
                        CustomerRow
                        parentCustomerRowByCustomerInvoice)
        {
            ...
        }

        public new InheritedInvoiceRow NewInvoiceRow()
        {
            ...
        }

        ...
    }

    ...
}
```

6.4 SIMPLIFICATION OF BUSINESS OBJECT LAYERS ■ 183

We first create a new indexer to return our new DataRow class to support returning our new DataRow, instead of the base class's indexer, which returns the base class. Next, we create new `AddInvoiceRow()` and `NewInvoiceRow()` methods (created in the Typed DataSet) to return our new DataRow class as well. Now our derived class should be type-safe and completely inherited.

Now that all the right plumbing is there, we can put our business logic in our DataRow class, as shown in Listing 6.18.

Listing 6.18: Adding Our Business Logic

```
public class InheritedTDS : FixedCustomerTDS
{
    ...

    public class InheritedInvoiceRow : InvoiceRow
    {
        ...

        public new DateTime InvoiceDate
        {
            get
            {
                return base.InvoiceDate;
            }
            set
            {
                if (value > DateTime.Today)
                {
                    throw new StrongTypingException(
                        "Invoice Date Cannot be in the future",
                        null);
                }
                else
                {
                    base.InvoiceDate = value;
                }
            }
        }
        ...
    }
}
```

```

    }
    ...
}

```

In this business logic, we wanted to check to make sure that whenever an invoice date is set, it is not set in the future. Putting this logic into the property makes sure users cannot set the invoice date incorrectly, but since the `DataRow` is ultimately the base class, we will need to make sure that users cannot set our invoice date by using the indexer (which would skip calling our property). One way we can do this is by overriding the indexer to make it read-only (see Listing 6.19).

Listing 6.19: *Protecting Our Business Logic*

```

public class InheritedTDS : FixedCustomerTDS
{
    ...

    public class InheritedInvoiceRow : InvoiceRow
    {
        ...

        // Only allow read by the indexer
        public new object this[int index]
        {
            get
            {
                return base[index];
            }
        }
        ...
    }

    ...
}

```

By overriding the indexer, users will have to use the properties to set the individual values. This way we do not have to duplicate our business logic or route the calls through our properties.

In our example, I derived from the invoice `DataTable` and the invoice `DataRow`. Our Typed `DataSet` also has type-safe classes for the customer's `DataTable` and `DataRow`. We did not derive from these because we did not need to add any business logic. So you can see that when you inherit from a Typed `DataSet` you do not need to derive from every `DataTable`, only the ones you want to add business logic into.

In this section we were able to derive from the Typed `DataSet` by manually editing the generated code. Because the generated code could change, it is generally a bad idea to edit it. To allow you to inherit without editing the generated code, Chris Sells and I have made a Visual Studio Add-in that will replace the standard Typed `DataSet` generator to make these changes for you in the generated code. You can download this code at my Web site (www.adoGuy.com/book/AGDataSetGenerator).

6.5 Conclusion

We can probably agree that type-safety is good and that by generating Typed `DataSets` we can reduce mistakes and misuse of our database. You have also learned how to generate Typed `DataSets` in Visual Studio .NET and from the command line, as well as how to customize the generation by using annotations.

Typed `DataSets` allow us to employ a new programming paradigm where we specialize generated classes to put business logic into our ADO.NET code. The code that Microsoft generates is problematic for using this paradigm, but we can get around this with a tool that is available on my Web site.

