CHAPTER 5

.NET data providers

The first part of this book provided a *very* high-level overview of the big pieces of ADO.NET. Starting with this chapter, part 2 will get closer to the details of the classes and show how to make this stuff work.

A lot of the more exciting aspects of ADO.NET relate to the use of the DataSet, which is the only topic of part 3. However, before getting to that, you need to understand how to do all the relatively boring, straightforward database stuff we all know and love.

This chapter explains the reasons for the existence of data providers and gives you a summary of the main classes in each provider.[1] In chapters 6 through 14 you will see these objects being used in real-world examples.

5.1 WHAT IS A DATA PROVIDER?

Quite simply, a *data provider* is the liaison between your code and the database. Rather than providing a single set of generic handlers for talking to databases, there is a data provider to talk to each different type of database/data-source (see figure 5.1).

[1] If you were using the first beta of .NET, data providers were called managed providers.

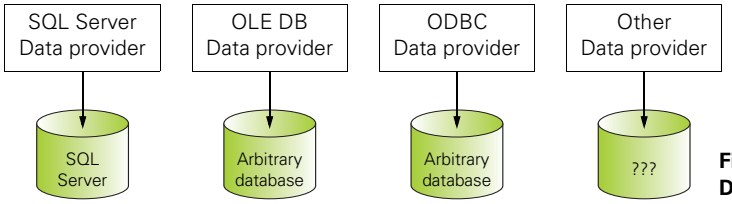

Figure 5.1
Data providers

I just told you that a custom data provider exists for each flavor of database—you may have noticed, however, that figure 5.1 shows a fairly slim set of data providers. Chalk this down to the fact that .NET is extremely new—the only data providers available today are those that Microsoft has built. The assumption is that each major database vendor will write its own custom data provider (strangely, Microsoft seems reticent about building its own high-performance Oracle or DB/2 data provider[2]).

Does this mean that you are stuck with using only SQL Server? Not at all. One of the data providers is a wrapper for OLE DB, Microsoft's previous major database access technology. Many companies have already written OLE DB providers, and using the ADO.NET OLE DB data provider gives you access to all of these legacy providers.

Microsoft has also released an ODBC data provider, although it doesn't ship with .NET. It allows access to the huge number of ODBC drivers that are out there. (ODBC is the previous, *previous* data access mechanism from Microsoft.)

I'll say a few words about each of the different providers.

5.1.1 SQL Server data provider

The SQL Server data provider is optimized for talking to Microsoft SQL Server and MSDE, the Desktop Edition of SQL Server. Because it is so specific, and it talks directly to SQL Server, it is considerably faster than the OLE DB data provider when talking to SQL Server.

The SQL Server data provider works with SQL Server version 7.0 or greater, although some functionality requires at least SQL Server 2000.

5.1.2 OLE DB data provider

The OLE DB data provider literally sits on top of OLE DB, and can be used to talk to virtually any data source that OLE DB can talk to. The advantage of this data provider is that it lets you use a large number of legacy drivers. The disadvantage is that it is generic, so it does not perform quite as well as a provider written specifically to a database engine.

The OLE DB data provider will not talk to all OLE DB data providers, because OLE DB is designed to talk to a large range of data (relational databases, OLAP,

[2] It has come to my attention that my sense of humor isn't always obvious to everyone. Just in case: yes, I am being ironic here!

LDAP, and so forth), whereas ADO.NET is much more focused on talking to relational data. You also cannot use the OLE DB provider that wraps ODBC, but you can use the ODBC data provider.

5.1.3 ODBC data provider

Although it didn't ship with the first version of .NET, the ODBC data provider gives you access to the myriad existing ODBC drivers available. There are many more ODBC drivers than OLE DB data providers, and frequently multiple different drivers are available for the same data source.

Appendix A discusses the installation and use of the ODBC data provider.

5.1.4 Other data providers

It is expected that other database vendors will produce data providers that talk specifically to their engines. In the next year or so, expect to see data providers for Oracle, DB/2, Sybase, and many other databases.

5.2 HOW ARE DATA PROVIDERS ORGANIZED?

Each data provider is in its own namespace and provides a full collection of objects that can be used directly. Most of the objects, though, are derived from a set of common interfaces.

5.2.1 Interfaces and namespaces

The approach of providing a set of common interfaces gives a lot of flexibility to the writers of data providers:

- Because most objects are derived from common interfaces, they can be used quite interchangeably and consistently. For example, if you mostly stick to using the interfaces in your code, you can *fairly* easily switch your code from using SQL Server to using Oracle (once an Oracle data provider is written).
- If a data provider needs to provide special functionality unique to it, it can just provide those specialized classes and methods. For example, SQL Server has a lot of native XML capabilities, so Microsoft has added methods to its versions of the objects to access these capabilities.

Each data provider is stored in its own namespace, which must be referenced to be used.

NAMESPACE A namespace is a logical grouping of classes and other supporting things like interfaces, enums, and so forth. .NET is broken into a large number of different namespaces, based on the functionality they contain. For example, the SQL Server data provider's code is contained in a namespace called `System.Data.SqlClient`.

Table 5.1 shows the namespaces that are important to ADO.NET.

Table 5.1 ADO.NET namespaces

Data provider	Namespace
Data provider interfaces	`System.Data`
SQL Server data provider	`System.Data.SqlClient`
OLE DB data provider	`System.Data.OleDb`
Common objects and utilities	`System.Data.Common`

When a company creates a new data provider, it is free to add its own classes and methods that are specific to the data source. However, all data providers are expected to have a certain set of objects that behave in a certain way. Of course, each new data provider will be in its own unique namespace.

5.3 STANDARD OBJECTS

Microsoft has created a set of interfaces that defines this basic set of objects, and each existing data provider implements these interfaces. For example, an interface called IDbConnection defines functionality for connecting to a database. For SQL Server, this interface is implemented by a class called SqlConnection, and for OLE DB, the interface is implemented by a class called OleDbConnection.

INTERFACE An interface is a definition for a set of properties and methods with no implementation. A class is said to *implement* an interface if it is derived from that interface. For this to happen, the class must contain all the methods and properties that the interface defines.

Although an object in C# can be derived from only one other object, it can be derived from, or *implement*, any number of interfaces.

Figure 5.2 shows the classes that should exist in every data provider. The diagram is generic in that it doesn't show the specific names of either the interfaces or the classes that implement those interfaces for each provider.

The following sections talk about the standard data provider classes. Among other things, for each object, we'll examine the interface and the specific classes that implement the interface. Each subsection will also explain, at a high level, the purpose for each class.

In chapters 6, 7, and 8, you will see these classes in use; the remaining chapters of part 2 will delve into the intricacies of the more important classes.

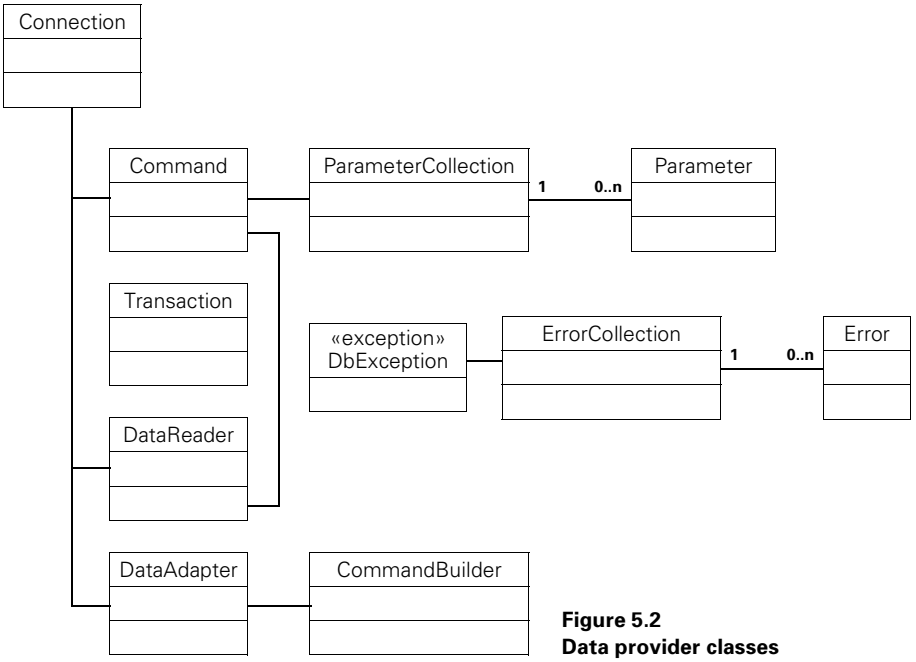

Figure 5.2
Data provider classes

5.3.1 The Connection interface and classes

Table 5.2 shows the classes that implement the connection for each of the providers, as well as the interface these classes implement. If you are working with just one type of data provider, you will rarely care about the interfaces; but if you want to write more generic code, they become more critical. (Chapter 8 talks in depth about writing generic code.)

Table 5.2 Connection implementations

Class	Provider	Interface
SqlConnection	SQL Server	IDbConnection
OleDbConnection	OLE DB	IDbConnection

A connection object represents a connection to the associated data source (SQL Server or any OLE DB data source). As you might assume, you need to have a connection before you can do anything useful with a data source.

The primary property on a connection is the connection string that tells it how to connect to the data source. An SQL Server connection string example might be as follows:

```
"server=ArF733;database=Test;user id=sa"
```

The arguments in this example specify that the SQL Server engine is running on a server called ArF733, the data is stored in a database called Test, and the user ID to log in with is *sa*—the system administrator's account. (As is all too common, this example doesn't include a password for the system administrator's account. Of course, you wouldn't do that on your server, would you?)

Probably the most important methods on a connection are

- `Open()`—Opens the connection
- `Close()`—Closes the connection

In addition, a number of properties provide information about the connection once it has been opened and about handling for transactions. (The next couple of chapters will show connection objects in use, and chapter 9 discusses the connection object and connection strings.)

Just about every example in this book relies on a connection object. You will see a direct example in chapter 6.

5.3.2 The Command interface and classes

Table 5.3 shows the classes and interface for the command object.

Table 5.3 Command implementations

Class	Provider	Interface
SqlCommand	SQL Server	IDbCommand
OleDbCommand	OLE DB	IDbCommand

If you want to do something with a data source—read data, change data, or just about anything else—you will almost always need to use a command object. The command object is used to send instructions to a data source.

A command has a couple of important properties:

- `Connection`—The connection object against which the command will be executed
- `CommandText`—The SQL command to be sent

There are several different methods for executing the SQL command, depending on the expected result. The two you will use most are:

- `ExecuteReader()`—Used for `Select` statements or any command expected to return a result set. This method returns a DataReader that can be used to move through the data in a forward-only, read-only manner.
- `ExecuteNonQuery()`—Used for statements that are not expected to return a result set, such as `Inserts`, `Updates`, and `Deletes`. This method returns the number of rows affected by the command.

Commands will be shown in use in the next couple of chapters, and chapter 10 goes into many of the gory details of the command object.

5.3.3 The ParameterCollection and Parameter interfaces and classes

For each provider, there is both a parameter implementation and a "collection of parameters" implementation (table 5.4).

Table 5.4 ParameterCollection and Parameter implementations

Class	Provider	Interface
SqlParameterCollection	SQL Server	IDataParameterCollection
SqlParameter		IDataParameter
OleDbParameterCollection	OLE DB	IDataParameterCollection
OleDbParameter		IDataParameter

A ParameterCollection holds the parameters that are associated with a particular command. Parameters are used to provide additional information to the database about SQL that is being executed.

There are two major reasons to use Parameters with SQL. The first and most common reason is to send arguments to stored procedures; second, parameters are also useful for certain types of efficiency issues.

Parameters and stored procedures

Stored procedures are basically little functions that exist inside the database server. Like regular functions, they can take parameters as arguments. You can pass these arguments as you would pass arguments to a regular function:

```
"MyStoredProcedure 'C2','Mr. Roberts'";
```

You can also bind values to the parameters directly:

```
Parameters["@Classroom"].Value = "C2";
Parameters["@Teacher"].Value = "Mr. Roberts";
```

This approach offers several advantages:

- It can be more efficient (although not always, depending on how often the procedure will be called).
- The data is passed as the native data type, rather than converted to text to be embedded in the SQL and then converted back to the original type by the database engine.
- Parameters can be used to retrieve data from the stored procedure as well as to send it.
- It is easy to specify some parameters and not others, without worrying about the order of the parameters.
- Because the name of the parameter is shown, it makes the code more readable.

Efficiency issues

Most commonly in SQL, when a statement is written, it contains all the information required to execute:

```
UPDATE Teachers SET Classroom='A1' WHERE Name='Mr. Biddick'
```

This statement will work fine for the specific teacher and classroom (Mr. Biddick, A1), but what happens if a number of classrooms need to be changed for a number of teachers? Of course, it would be possible to build a new string for each case and execute it separately, but that approach has a couple of problems:

- It is inefficient to build strings (especially with more complex examples), reset all the objects, and execute everything from scratch.
- Many databases are smart enough to remember how to do specific tasks so that they are faster the next time. This is usually referred to as an *execution plan*, and you can think of it as a compiled version of a query.

The problem with an execution plan is that it is generally based on the exact text of the SQL. Therefore, the following two strings would be considered completely different statements:

```
UPDATE Teachers SET Classroom='A1' WHERE Name='Mr. Biddick'
UPDATE Teachers SET Classroom='B2' WHERE Name='Ms. Fortune'
```

By using parameters, you can put in *placeholders* for the data that will change, and then specify the appropriate values:

```
UPDATE Teachers SET Classroom=? WHERE Name=?
```

Each `?` will be replaced with a value at runtime when binding takes place.[3] The following code can be used to specify the values for each parameter from the previous example:

```
Parameters[0].Value = "A1";
Parameters[1].Value = "Mr. Biddick";
```

The same update can then be executed over and over, just changing the values for the parameters.

Chapter 12 goes into depth about using parameters, and chapter 13 talks about using parameters with stored procedures.

[3] The format for parameters is different for SQL Server and OLE DB. The question mark (`?`) is used for OLE DB. SQL Server uses named parameters.

5.3.4 The Transaction interface and classes

Unlike many previous data-access technologies, ADO.NET has objects to represent transactions (table 5.5).

Table 5.5 Transaction implementations

Class	Provider	Interface
SqlTransaction	SQL Server	IDbTransaction
OleDbTransaction	OLE DB	IDbTransaction

Normally, when an SQL command is executed through ADO.NET, the effects of the command take place immediately. Sometimes, though, it is highly desirable to tie together several different commands and have them all succeed or fail as a unit. Consider the following example:

```
UPDATE Inventory SET Quantity=Quantity-1 WHERE Item='Wdgt'
INSERT INTO Orders (Item,Quantity) VALUES ('Wdgt',1)
```

If the first command completes but the second one does not, then the inventory will show an incorrect count, because the order will not have gone through.

By wrapping these commands in a transaction, they can be treated as a single, atomic event. The transaction can then be *committed*, in which case the results of both commands will be made permanent; or the transaction can be *rolled back*, in which case neither of the commands will appear to have happened.

There is a class to represent a transaction, rather than just some methods on the connection class, because it allows multiple transactions to be handled simultaneously. In theory, you could have one transaction that handles one set of operations while another unrelated transaction runs independently. Unfortunately, though, that approach is not really supported by the underlying databases—the databases cannot handle independent transactions on the same connection. However, in the future, the databases will support this capability, and .NET already has a design that can support it when it appears.

Chapter 14 discusses transactions. Chapter 33 talks about transactions that involve more than one data source or other objects (distributed transactions).

5.3.5 The DataReader interfaces and classes

A DataReader is used for reading results from an SQL statement. The DataReader objects implement more than one interface (table 5.6).

Table 5.6 DataReader implementations

Class	Provider	Interface
SqlDataReader	SQL Server	IDataReader, IDataRecord
OleDbDataReader	OLE DB	IDataReader, IDataRecord

SQL queries can be broken roughly into two general categories—those that return a result set and those that do not (profound, eh?). For commands that do not return a result set (`Insert`, `Update`, `Delete`, and so forth) the most you can expect back is whether the command succeeded, and maybe a count of the number of records affected.

For commands that *do* return a result set (`Select` and certain stored procedures), there needs to be a way of looking at the results. The DataReader is the ADO.NET mechanism for quickly stepping through a result set.

A DataReader has the ability to step through the result set in a *forward-only* manner. That means it can't go backward or jump arbitrarily around in the data. For each record, methods are available to return the data in each column.

DataReaders are designed to be fast—ask for data, read the data, close the connection. Although the methods on a DataReader only allow for a single record to be read at a time, it is highly likely that the underlying implementation reads much bigger blocks of data at a time for efficiency. DataReaders are also read-only, so they don't need to keep track of where data came from or how to put it back.

DataReaders will be used in the next several chapters to look at returned data, and will be discussed in detail in chapter 11.

5.3.6 The DataAdapter interface and classes

Those of you who have read a little about ADO.NET and have heard of the DataSet might be surprised that the DataSet so far has not featured in any of the discussions about data providers. That is the case because the DataSet is specifically designed to be "data source agnostic." Once you have data in a DataSet, you can move it to other tiers of your application (via remoting), tie it to user interface objects (such as data tables), and generally work with your data in any way you need.

As useful as this capability sounds, the DataSet is designed to work with relational data that comes from a database. It would be very annoying if you had to manually transfer data from the DataSet to the database code just to achieve this functionality.

That is where the DataAdapter classes come in. Table 5.7 shows the DataAdapter classes and interface.

Table 5.7 DataAdapter implementations

Class	Provider	Interface
SqlDataAdapter	SQL Server	IDbDataAdapter
OleDbDataAdapter	OLE DB	IDbDataAdapter

In most previous data access technologies, the way data was accessed was tied tightly to the way the data was manipulated. For example, in ADO, a Recordset holds data and can directly send changes to the database when the user changes data within the Recordset.

That sounds very object-oriented—you tell the object to do something, and it does it. The problem is that this approach is very *tightly coupled.* It assumes that the data connection is always available and, for that matter, that the data is coming from a database.

Separating the part of the code responsible for talking to the database from the part of the code used for manipulating data offers a number of advantages:

- It is no longer necessary to maintain a connection while the data manipulation is taking place, which means the connection can be used elsewhere.
- The data manipulation can take place on a different tier of your application (possibly a physically different computer) than the connection to the database, by transferring the information to be manipulated (as with a classic three-tier application).
- The data manipulation code is no longer tied to just database access—it becomes a generic mechanism for manipulating data.
- You can use one set of data access mechanisms for reading the data and a different mechanism for writing it later. You can, for example, read the data from one database and write it to a different database.
- Because the data manipulation code does not have data access code, and the data access code does not have data manipulation code, both sets of code become simpler.

ADO started to address this capability with Disconnected Recordsets. ADO.NET takes the concept further. The two different objects are

- *The DataAdapter*—Responsible for obtaining the data and, later, for writing changes to the database.
- *The DataSet*—The object that allows for manipulation of the data. Not only does it have capabilities for adding, removing, and changing data, but it is also designed to be easily transportable to a different tier via remoting mechanisms.

The DataAdapter contains four important properties, each of which is really nothing more than a command object. These four properties are:

- `SelectCommand`—Used to select the data to use
- `InsertCommand`—Used to insert new data
- `UpdateCommand`—Used to update existing data
- `DeleteCommand`—Used to remove data

Most of these command objects can be automatically generated by ADO.NET via the use of a CommandBuilder (discussed next), or can be custom provided to handle special cases, use stored procedures, and so forth.

You can load data into a DataSet via the `Fill()` method on a DataAdapter. The `Update()` method takes the changes in a DataSet and, using the appropriate `Insert`, `Update`, and `Delete` commands, writes the data back to the data source.

The DataSet and the DataAdapter are the topic of the chapters in part 3 of this book.

5.3.7 CommandBuilder classes

In the previous section, I talked about the fact that you can manually specify the commands to use for `Inserts`, `Updates`, and `Deletes` for a DataAdapter. That is all well and good, but what if you just want the system to handle the simple cases for you? That is where the CommandBuilders come in (table 5.8).

Table 5.8 CommandBuilder implementations

Class	Provider	Interface
SqlCommandBuilder	SQL Server	n.a.
OleDbCommandBuilder	OLE DB	n.a.

You may notice that, unlike the other classes discussed so far, the CommandBuilders do not implement an interface. That is because they are really just utility objects that may or may not be provided by any particular data provider, or may be implemented in a substantially different way.

The CommandBuilders are utility objects especially designed to work with a DataAdapter. The previous section explained that most of the command objects for a DataAdapter can be automatically created; this is done using the CommandBuilder.

Because the DataAdapter is designed to be flexible, it does not have built-in mechanisms for any of the commands. But Microsoft knows that a great many DataAdapters will use fairly straightforward `Selects`, and fairly straightforward `Insert`, `Update`, and `Delete` commands; so, the company provided the CommandBuilder to avoid developers' having to generate this common code.

The CommandBuilder cannot do several tasks:

- It cannot generate a `Select` statement automatically—there is no way for the CommandBuilder to know what data is desired.
- It cannot generate `Insert`, `Update`, or `Delete` statements for overly complex queries. For example, if the query contains a join, then the CommandBuilder cannot be used.
- It cannot take advantage of special knowledge about the data, or use stored procedures and so forth. Any special behavior must be implemented by the developer.

What it can do, though, is speed up the use of DataAdapters for the most common cases. It is important to know, though, that the commands generated by the CommandBuilders are not necessarily efficient. CommandBuilders are discussed in part 3, which focuses on the use of the DataSet.

5.3.8 The DbException, ErrorCollection, and Error classes

The exception-handling and error-handling in ADO.NET is specific to the data provider (table 5.9).

Table 5.9 Exception and error implementations

Class	Provider	Interface
SqlException	SQL Server	n.a.
SqlErrorCollection		
SqlError		
OleDbException	OLE DB	n.a.
OleDbErrorCollection		
OleDbError		

Once again, you will notice that there are no general interfaces here. Although I understand the reason—the error information returned by the different providers is quite different—I still wish Microsoft had created even a simple base class for these objects. At least a couple of properties are basically the same from any provider, and a base class would have made coding generically somewhat simpler.

.NET generally uses exceptions to indicate errors, and ADO.NET follows this standard. When a database error occurs, the appropriate exception is thrown. Issues that can throw an exception include:

- *Incorrect use of a database object*—For example, attempting to execute a command before a connection has been made to the database
- *Invalid SQL*—SQL that is either illegal or references missing tables or columns
- *Missing parameters*—SQL that references parameters that are not specified
- *Permission problems*—SQL that attempts an operation the client does not have rights to perform

Once an exception has been caught, you can step through the error collection and look at each error in turn. The error objects differ from SQL Server to OLE DB, but information they generally contain includes:

- *Message*—A short description of the error
- *Number/state*—A unique identifier for the type of error

Although no specific chapter discusses error handling in detail, you'll see error handling in several examples throughout the rest of the book.

5.4 SUMMARY

This chapter has presented something of a whirlwind tour of the major classes within a data provider. It should be enough to make you comfortable with the objects you will encounter in the next couple of chapters, which provide a hands-on look at using these classes.

Those of you who have a lot of experience using previous data access mechanisms, such as ADO, probably feel that not a lot is new here, beyond some terminology

changes. Especially at this level, that is fairly true; much of what you read in this chapter and will see in the next two chapters will be little more than a chance to pick up the ADO.NET syntax.

One good thing is that the various classes are nicely segmented and simple. In particular, because each data provider has its own set of classes for everything, you don't have to wade through irrelevant methods or work with mechanisms made complicated by having to support a weird case. At the same time, there *are* interfaces that allow for code to be written generically.

You will learn many other useful things about ADO.NET throughout the rest of the book. Also, Microsoft claims that ADO.NET is faster than ADO. For example, the .NET SQL Server data provider has shown itself to be 10 to 20 percent faster than using ADO to access SQL Server via OLE DB!

So much for the theory. In the next chapter, we will finally get around to using ADO.NET to do something.