CHAPTER 3

ADAPTER

An object is a *client* if it needs to call your code. In some cases, client code will be written after your code exists and the developer can mold the client to use the interfaces of the objects that you provide. In other cases, clients may be developed independently of your code. For example, a rocket simulation program might be designed to use rocket information that you supply, but such a simulation will have its own definition of how a rocket should behave. In such circumstances, you may find that an existing class performs the services that a client needs but with different method names. In this situation, you can apply the ADAPTER pattern. The intent of ADAPTER is to provide the interface that a client expects while using the services of a class with a different interface.

Adapting to an Interface

When you need to adapt your code, you may find that the client developer planned well for such circumstances. This is evident when the developer provides an interface that defines the services that the client code needs, as the example in Figure 3.1 shows. A client class makes calls to a `RequiredMethod()` method that is declared in an interface. You may have found an existing class with a method that can fulfill the client's needs, with a name such as `UsefulMethod()`. You can adapt the existing class to meet the client's needs by writing a class that extends `ExistingClass`, implements `RequiredInterface`, and overrides `RequiredMethod()` so that it delegates its requests to `UsefulMethod()`.

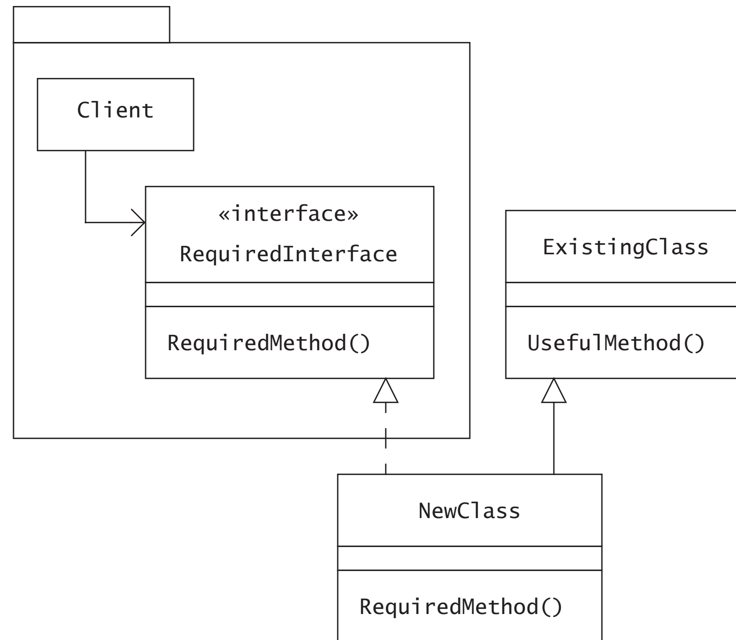

Figure 3.1 When a developer of client code thoughtfully defines the client's needs, you may be able to fulfill the interface by adapting existing code.

The `NewClass` class in Figure 3.1 is an example of ADAPTER. An instance of this class is an instance of `RequiredInterface`. In other words, the `NewClass` class meets the needs of the client.

For a more concrete example, suppose you are working with a package that simulates the flight and timing of rockets such as those you manufacture at Oozinoz. The simulation package includes an event simulator that explores the effects of launching several rockets, along with an interface that specifies a rocket's behavior. Figure 3.2 shows this package.

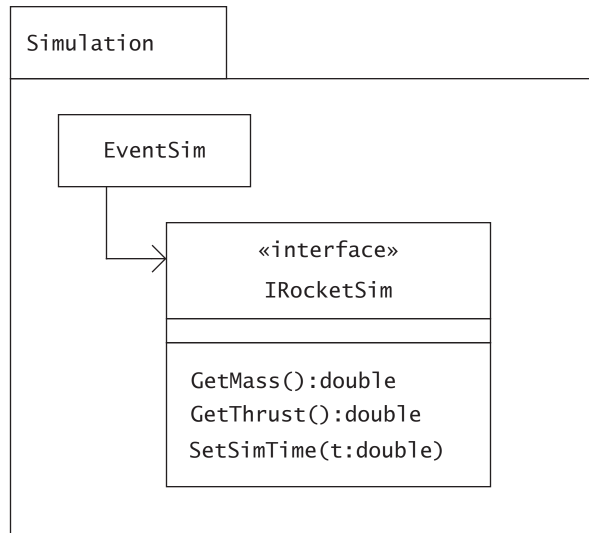

Figure 3.2 The `Simulation` package clearly defines its requirements for simulating the flight of a rocket.

Suppose that at Oozinoz you have a `PhysicalRocket` class that you want to plug into the simulation. This class has methods that supply, approximately, the behavior that the simulator needs. In this situation, you can apply ADAPTER, creating a subclass of `PhysicalRocket` that implements the `IRocketSim` interface. Figure 3.3 partially shows this design.

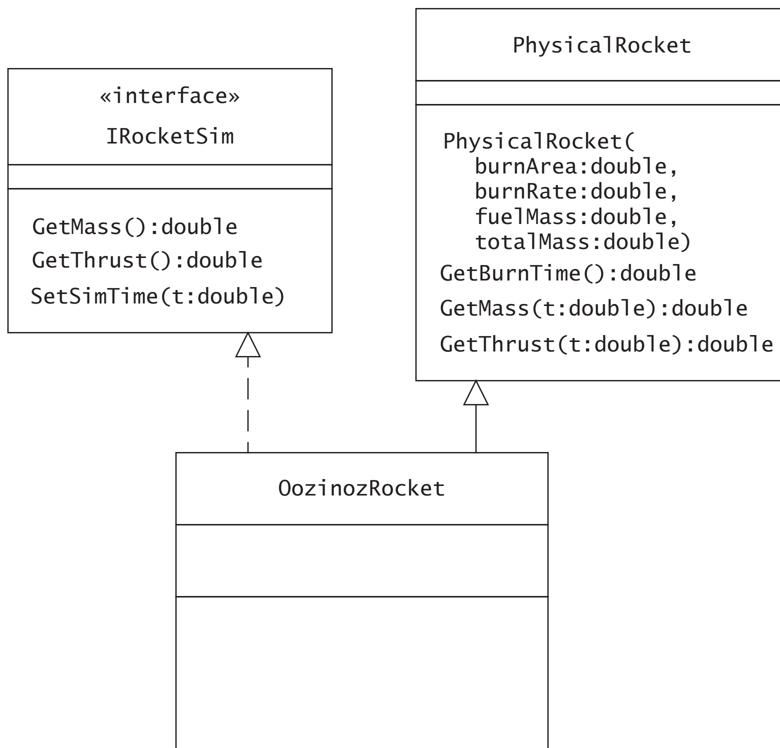

Figure 3.3 When complete, this diagram shows the design of a class that adapts the `Rocket` class to meet the needs of the `IRocketSim` interface.

The `PhysicalRocket` class has the information that the simulator needs, but its methods do not exactly match those that the simulation declares in the `IRocketSim` interface. Most of the differences occur because the simulator keeps an internal clock and occasionally updates simulated objects by calling a `SetSimTime()` method. To adapt the `PhysicalRocket` class to meet the simulator's needs, an `OozinozRocket` object can maintain a `_time` instance variable that it can pass to the methods of the `PhysicalRocket` class as needed.

Challenge 3.1

Complete the class diagram in Figure 3.3 to show the design of an `OozinozRocket` class that lets a `PhysicalRocket` object participate in a simulation as an `IRocketSim` object.

A solution appears on page 350.

The code for `PhysicalRocket` is somewhat complex as it embodies the physics that Oozinoz uses to model a rocket. However, it is exactly that logic we want to reuse without reimplementing. The `OozinozRocket` class simply translates calls to use its superclass's methods. The code for this new subclass will look something like:

```
public class OozinozRocket : PhysicalRocket, IRocketSim
{
    private double _time;
    public OozinozRocket(
        double burnArea, double burnRate,
        double fuelMass, double totalMass)
        : base (burnArea, burnRate, fuelMass, totalMass)
    {
    }
    public double GetMass()
    {
       // challenge!
    }
    public double Thrust()
    {
        // challenge!
    }
    public void SetSimTime (double time)
    {
        _time = time;
    }
}
```

Challenge 3.2

Complete the code for the `OozinozRocket` class, including methods `GetMass()` and `GetThrust()`.

A solution appears on page 351.

When a client defines its expectations in an interface, you can apply ADAPTER by supplying a class that implements a provided interface and that subclasses an existing class. You may also be able to apply ADAPTER

even if no interface exists to define a client's expectations. In this situation, you must use an "object adapter."

Class and Object Adapters

The designs in Figures 3.1 and 3.3 show *class adapters* that adapt through subclassing. In a class adapter design, the new adapter class implements the desired interface and subclasses an existing class. This approach will not always work, particularly when the set of methods that you need to adapt is not specified in a C# interface. In such a case, you can create an *object adapter,* an adapter that uses delegation rather than subclassing. Figure 3.4 shows this design.

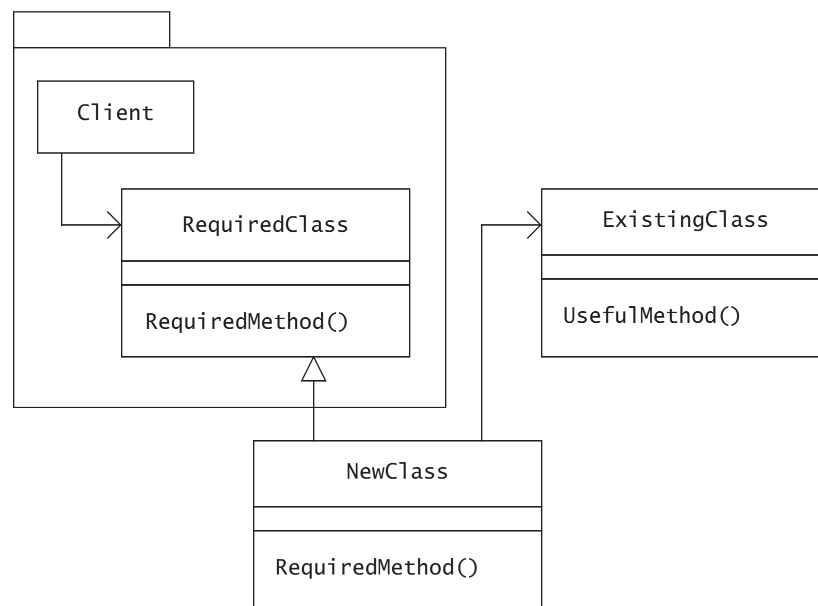

Figure 3.4 You can create an object adapter by subclassing the class that you need, fulfilling the required methods by relying on an object of an existing class.

The `NewClass` class in Figure 3.4 is an example of ADAPTER. An instance of this class is an instance of the `RequiredClass` class. In other words, the `NewClass` class meets the needs of the client. The `NewClass` class can adapt the `ExistingClass` class to meet the client's needs by using an instance of `ExistingClass`.

For a more concrete example, suppose that the simulation package worked directly with a `Skyrocket` class, without specifying an interface to define the behaviors the simulation needs. Figure 3.5 shows this class.

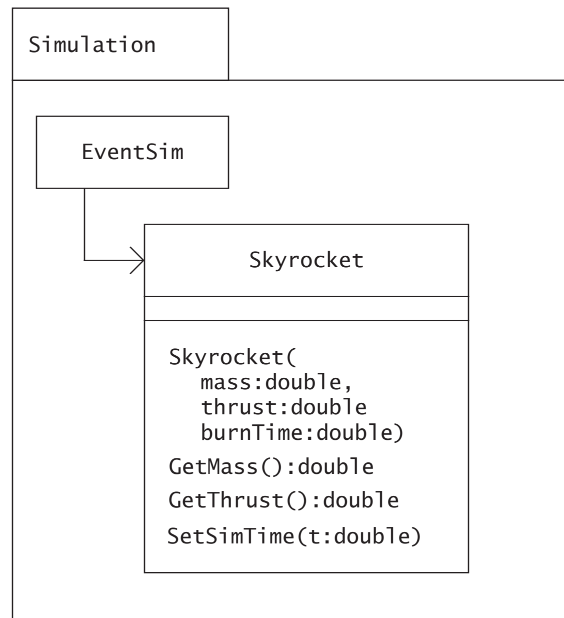

Figure 3.5 In this alternative design, the `Simulation` package does not specify the interface it needs for modeling a rocket.

The `Skyrocket` class uses a fairly primitive model of the physics of a rocket. For example, it assumes that the rocket is entirely consumed as its fuel burns. Suppose that you want to apply the more sophisticated physical model that the Oozinoz `PhysicalRocket` class uses. To adapt the logic in the `PhysicalRocket` class to the needs of the simulation, you can create an `OozinozSkyrocket` class as an object adapter that subclasses `Skyrocket` and that uses a `PhysicalRocket` object, as Figure 3.6 shows.

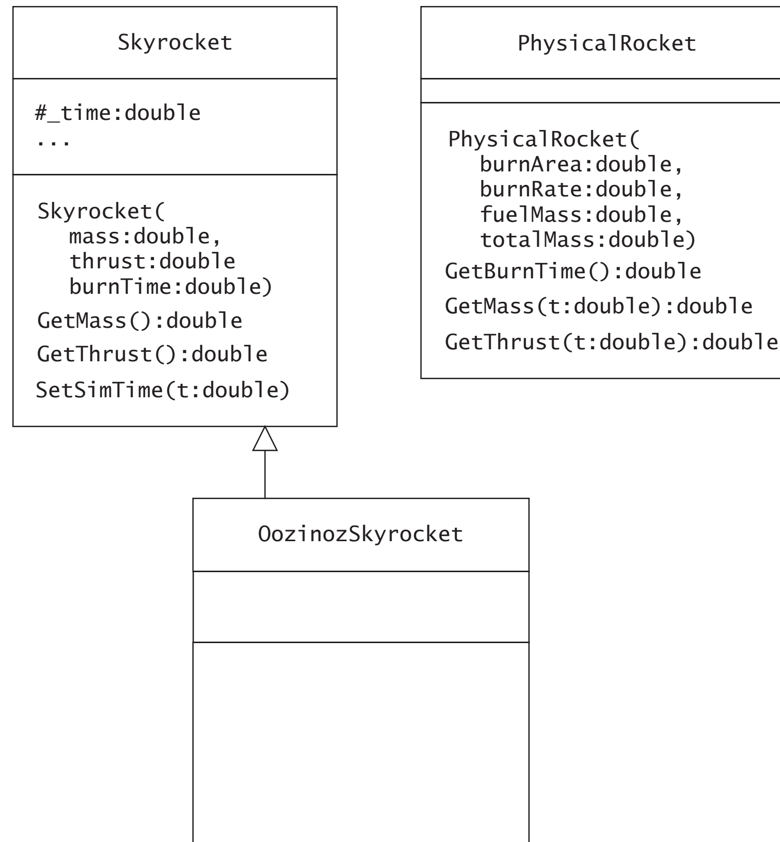

Figure 3.6 When complete, this diagram shows an object adapter design that uses information from an existing class to meet the needs that a client has of a `Skyrocket` object.

Notice that the `OozinozSkyrocket` class subclasses from `Skyrocket`, not `PhysicalRocket`. This will allow an `OozinozSkyrocket` object to serve as a substitute wherever the simulation client needs a `Skyrocket` object. The `Skyrocket` class supports subclassing by making its `_time` variable `protected` (as shown in the UML diagram) and by making its methods `virtual` (not shown in the diagram).

Challenge 3.3

Complete the class diagram in Figure 3.6 to show a design that allows `OozinozRocket` objects to serve as `Skyrocket` objects.

A solution appears on page 351.

The code for the `OozinozSkyrocket` class might be as follows:

```
public class OozinozSkyrocket : Skyrocket
{
    private PhysicalRocket _rocket;
    public OozinozSkyrocket(PhysicalRocket r) :
        base (r.GetMass(0), r.GetThrust(0), r.GetBurnTime())
    {
        _rocket = r;
    }
    public override double GetMass()
    {
        return _rocket.GetMass(_simTime);
    }
    public override double GetThrust()
    {
        return _rocket.GetThrust(_simTime);
    }
}
```

The `OozinozSkyrocket` class lets you supply an `OozinozSkyrocket` object anywhere the simulation package requires a `Skyrocket` object. In general, object adapters partially overcome the problem of adapting an object to an interface that is not expressly defined.

Challenge 3.4

Name one reason why the object adapter design that the `OozinozSkyrocket` class uses may be more fragile than a class adapter approach.

Solutions appear on page 352.

The object adapter for the `Skyrocket` class is a more dangerous design than the class adapter that implements the `IRocketSim` interface. But we should not complain too much. At least the `Skyrocket` designer marked the methods as `virtual`. Suppose that this were not the case. Then the `OozinozSkyrocket` class could not override the `GetMass()` or `GetThrust()` methods. A subclass cannot force adaptability onto its superclass.

The best way to plan for adaptation is to define the needs of a client program in an interface. If you do not foresee a specific type of adaptation, you may still want other developers to be able to create object adapters for your class. In this case, place a `virtual` modifier on any methods that you want to let subclasses override.

Adapting Data in .NET

If you search for "adapter" in the online help for Visual Studio .NET, you will find that almost all the results relate to adapting database data. This is not too surprising, because a major goal of an n-tier architecture is to define how data is represented in each tier: in persistent storage, in business objects, and in visual presentations. An architecture must also supply mechanisms for transforming data between these representations, as we will have frequent need to adapt data in one tier to the meet the needs of another tier.

Although the .NET Framework Class Libraries provide ample support for adapting data to the needs of different architectural layers, not all data adapters are examples of the ADAPTER pattern. It is useful to look at a data adapter example and then ask whether ADAPTER plays a role or could play a role. Using the .NET FCL, it is easy to create an adapter that can take a structured query language (SQL) query and extract database data. The Oozinoz `DataServices` class encapsulates this adaptation as a service as follows:

```
using System;
using System.Data;
using System.Data.OleDb;
//...
namespace DataLayer
{
    public class DataServices
    {
        //...
        public static OleDbDataAdapter
            CreateAdapter (string select)
        {
            return new OleDbDataAdapter(
                select, CreateConnection());
        }
        //...
    }
}
```

The static `CreateAdapter()` method returns an "adapter" of type `OleDbDataAdapter` that contains the results of a SQL select statement. One of the most useful methods that the `OleDbDataAdapter` class supports is `Fill()`, a method that pushes database data from the adapter object into a `DataSet`

object. An instance of the `DataSet` class is essentially an in-memory relational database (sans engine) that houses tables and their relationships. Several graphical control classes, such as the `DataGrid` class, can extract data from a `DataSet` object. The `OldDbDataAdapter`, `DataSet`, and `DataGrid` classes collaborate to make it easy to wire together applications that whisk data along from a database, through a dataset, and into a visual representation.

To see a data adapter in action at Oozinoz, we need first to take a quick look at the Oozinoz `UI` utility class. This class provides several standard graphical user interface (GUI) objects, including a standard font and a standard `DataGrid` object, as the following code shows:

```
using System;
using System.Drawing;
using System.Windows.Forms;
namespace UserInterface
{
    public class UI
    {
        public static readonly UI NORMAL = new UI();
        protected Font _font =
            new Font("Book Antiqua", 18F);
        public virtual Font Font
        {
            get
            {
                return _font;
            }
        }
        //...
        public virtual DataGrid CreateGrid()
        {
            DataGrid g = new DataGrid();
            g.Dock = DockStyle.Fill;
            g.CaptionVisible = false;
            return g;
        }
    }
}
```

Using the `UI` class along with the `OldDbDataAdapter`, `DataSet`, and `DataGrid` classes, a short program can marry a database adapter to a data grid to display a database table as follows:

```
using System.Windows.Forms;
using System.Data;
using System.Data.OleDb;
using DataLayer;
using UserInterface;
public class ShowAdapter : Form
{
    public ShowAdapter()
    {
        DataSet d = new DataSet();
```

```
            string s = "SELECT * FROM Rocket";
            OleDbDataAdapter a = DataServices.CreateAdapter(s);
            a.Fill(d, "Rocket");
            a.Dispose();

            DataGrid g = UI.NORMAL.CreateGrid();
            g.SetDataBinding(d, "Rocket");
            Controls.Add(g);

            Text = "All My Rockets";
            Font = UI.NORMAL.Font;
        }
        static void Main()
        {
            Application.Run(new ShowAdapter());
        }
    }
```

This program creates an `OleDbDataAdapter` object that reads all the data in the `Rocket` table in the database. (The `Rocket` table is actually a "query" in the `oozinoz.mdb` Microsoft Access database. You can download this database and all the files that go with this book from www.oozinoz.com. See Appendix C, "Oozinoz Source," for help with obtaining the source.)

The adapter's `Fill()` method creates a `DataTable` object within the dataset and names the table `"Rocket"`. The program then releases its database resources with a call to `Dispose()` and creates a `DataGrid` object as the only control in the form. The `SetDataBinding()` method causes the grid to display data from the `Rocket` table within the supplied dataset.

Running this program produces the display shown in Figure 3.7.

All My Rockets

Name	Mass	Price	Apogee	Thrust
Biggie	1.5	29.95	820	110
Byebaby	0.3	13.95	75	30
Heightful	0.5	33.95	300	22
Jsquirrel	1	25.95	3000	52
Mach-it	0.35	22.95	1000	42
Orbit	1.25	29.95	5000	65
Pocket	0.01	2.95	12	3
Shooter	0.02	3.95	50	6
Sock-it	0.15	9.95	100	15
Sprocket	0.02	3.95	50	10

Figure 3.7 A few lines of C# code can produce this presentation of a database table's contents.

This example shows a flow of data from database through to presentation, but does not show the ADAPTER pattern at play. If ADAPTER were present, we would see an interface that defines the `DataGrid` class's needs (for a class adapter) or we would see subclasses of `DataGrid` (for an object adapter).

The lack of ADAPTER in this example does not imply that the design is inflexible. In fact, the logic in the `SetDataBinding()` method accepts many different types of arguments. A program can pass this method a `DataSet` instance, a `DataTable` instance, a `DataView` instance, a `DataViewManager` instance, or an instance of any class that implements the `IListSource` or `IList` interfaces. The method can also accept a one-dimensional array. Flexible indeed! But, rather than handling so many different sources of data, the `DataGrid` class might actually be more flexible if it defined its expectations in an interface. Figure 3.8 shows this approach, along with a class that adapts the interface to an instance of `IList`.

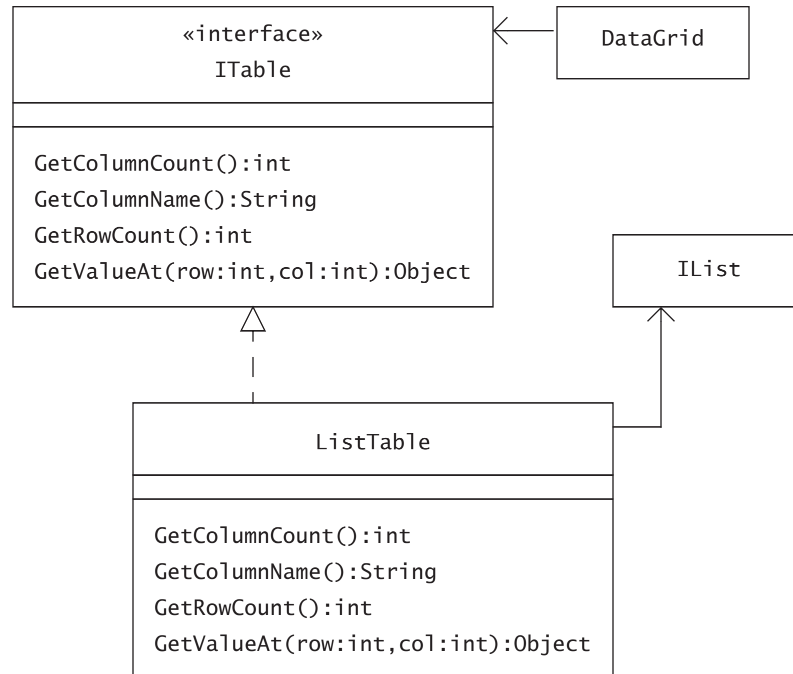

Figure 3.8 This design uses an adapter to adapt a source object's data to the needs of a `DataGrid` object.

Note that the design in Figure 3.8 is merely a proposal: The existing FCL classes are not laid out this way and there is no `ITable` interface in the FCL. The design suggests that the developers of the `DataGrid` class might supply an `ITable` interface to define the `DataGrid` class's needs. Then, instead of building support for `IList` objects into the `DataGrid` class's `SetDataBinding()` method, a `ListTable` class could adapt a list's data to appear as a table.

Challenge 3.5

The `DataGrid` class accepts many different data types as its data source instead of specifying its requirements in an interface. List two benefits of an ADAPTER design that would specify an `ITable` interface and that would provide adapter classes for different data sources.

A solution appears on page 353.

As data flows from persistent storage through business layers and into presentation code, there are often opportunities to adapt a data source to meet the needs of a data consumer. The ADAPTER pattern is not too prevalent in .NET, and a greater presence would arguably slim down control classes and provide more flexibility in how adaptation occurs.

Summary

The ADAPTER pattern lets you use an existing class to meet a client class's needs. When a client specifies its requirements in an interface, you can usually create a new class that implements the interface and subclasses an existing class. This approach creates a *class adapter* that translates a client's calls into calls to the existing class's methods.

When a client does not specify the interface it requires, you may still be able to apply ADAPTER, creating a new subclass class of the client that uses an instance of the existing class. This approach creates an *object adapter* that forwards a client's calls to an instance of the existing class. This approach can be dangerous, especially if you don't (or perhaps can't) override all the methods that the client might call.

Data flow in the .NET architecture provides many examples of adaptation, but few examples of the ADAPTER pattern. This is arguably a missed opportunity that occurs when a class such as `DataGrid` does not define its needs in an interface. When you architect your own systems, consider the power and flexibility that you and other developers can derive from an architecture that uses ADAPTER to advantage.