CHAPTER 7

Callback Interfaces, Delegates, and Events

Up to this point in the text, every sample application you have developed added various bits of code to Main(), which (in some way or another) sent messages to a given object. However, you have not yet examined how an object can talk back to the entity that created it. In most programs, it is quite common for objects in a system to engage in a “two-way conversation” through the use of events, callback interfaces, and other programming constructs. To prime the pump, I begin this chapter by examining how custom interfaces may be used to enable callback functionality (which may ring a bell if you are a COM developer).

The bulk of this chapter, however, examines various techniques provided by C# and the .NET Framework that enable the objects in your system to engage in bidirectional communications. First, you learn about the C# “delegate” keyword, which is little more than an object that “points to” other method(s) it is able to make calls on when told to do so.

Once you learn how to create and manipulate delegates (both synchronously and asynchronously), you then investigate the .NET event protocol, which is based on the delegation model. Although one typically views events within the context of a GUI-based application, this chapter will illustrate how non-GUI agents are able to fire events to their invokers (as shown later during our investigation of Windows Forms and ASP.NET, the process is identical to that of firing and handling GUI-centric events).

Understanding Callback Interfaces

As you have seen in the previous chapter, interfaces can be used to define common behaviors supported by various types in your system. In addition to using interfaces to establish polymorphic behaviors, interfaces are also commonly used as a callback mechanism. COM programmers may already be familiar with the notion of defining and implementing callback interfaces. This technique allows a COM client to receive events from a coclass using a custom COM interface, and is often used to bypass the overhead imposed by the official COM connection point architecture.

To illustrate the use of callback interfaces under .NET, let’s update the now familiar Car type to inform the caller when it is about to explode (the current speed is 10 miles below the maximum speed), and has exploded (the current speed is at or above the maximum speed). These events will be represented by the following custom interface:
// The callback interface.
public interface IEngineEvents
{
    void AboutToBlow(string msg);
    void Exploded(string msg);
}

Event interfaces are not typically implemented directly by the client executable, but
rather by a helper sink object, upon which the sender of the events (the Car type in this
case) will make calls. Assume the client-side sink class is called CarEventSink. When
the Car type sends the event notification to the sink, it will simply print out the incoming
messages to the console:

// Car event sink.
public class CarEventSink : IEngineEvents
{
    private string name; // Diagnostic member to identify sink.
    public CarEventSink(){}
    public CarEventSink(string sinkName)
    { name = sinkName; }
    public void AboutToBlow(string msg)
    { Console.WriteLine("{0} reporting: {1}", name, msg); }
    public void Exploded(string msg)
    { Console.WriteLine("{0} reporting: {1}", name, msg); }
}

Now that you have a sink object that implements the event interface, your next task
is to pass a reference to this sink into the Car type. The Car holds onto the reference,
and makes calls back on the sink when appropriate. In order to allow the Car to obtain
a reference to the sink, you can assume some method has been added to the default
public interface. In keeping with the COM paradigm, let's call this method Advise().
When the object user wishes to detach from the event source, it may call another
custom method on the Car type (Unadvise() in COM-speak).

Furthermore, in order to allow the call to register multiple event sinks, let's assume that
the Car maintains an ArrayList to represent each outstanding connection (analogous to the
array of IUnknown* types used with classic COM connection points). Here are the
relevant updates:

// This Car does not make any use of the
// C# delegate or event keyword, but can
// still send out events to the caller.
public class Car
{
    // The set of connected sinks.
    ArrayList itfConnections = new ArrayList();
    // Attach or disconnect from the source of events.
    public void Advise(IEngineEvents itfClientImpl)
    { itfConnections.Add(itfClientImpl); }
    public void Unadvise(IEngineEvents itfClientImpl)
    { itfConnections.Remove(itfClientImpl); }
    ...
}