Decouple Components With LCE

Loosely Coupled Events (LCE) simplifies code maintenance by letting subscribers and publishers interact through .NET.

by Juval Löwy

Technology Toolbox

- ❑ VB.NET
- ☑ C#
- ❑ SQL Server 2000
- ❑ ASP.NET
- ❑ XML
- ❑ VB6

Decoupling subscribers from publishers yields easier long-term maintenance, faster time to market, and more robust applications. Of course, doing this efficiently requires that you be able to add and remove subscribers without changing the publisher's code.

I'll explain the publish/subscribe model in detail, then show you how to decouple your own apps in .NET using Loosely Coupled Events (LCE). LCE lets subscribers subscribe to types of events, so when publishers change, subscriber code doesn't, and vice versa. LCE also lets you decouple the subscriber/publisher lifeline, so subscribers don't need to be running to receive the event. Instead of delivering an event directly, you deliver it to .NET, which delivers it to the subscribers.

An object provides services to clients in a component-oriented program by letting clients invoke meth-

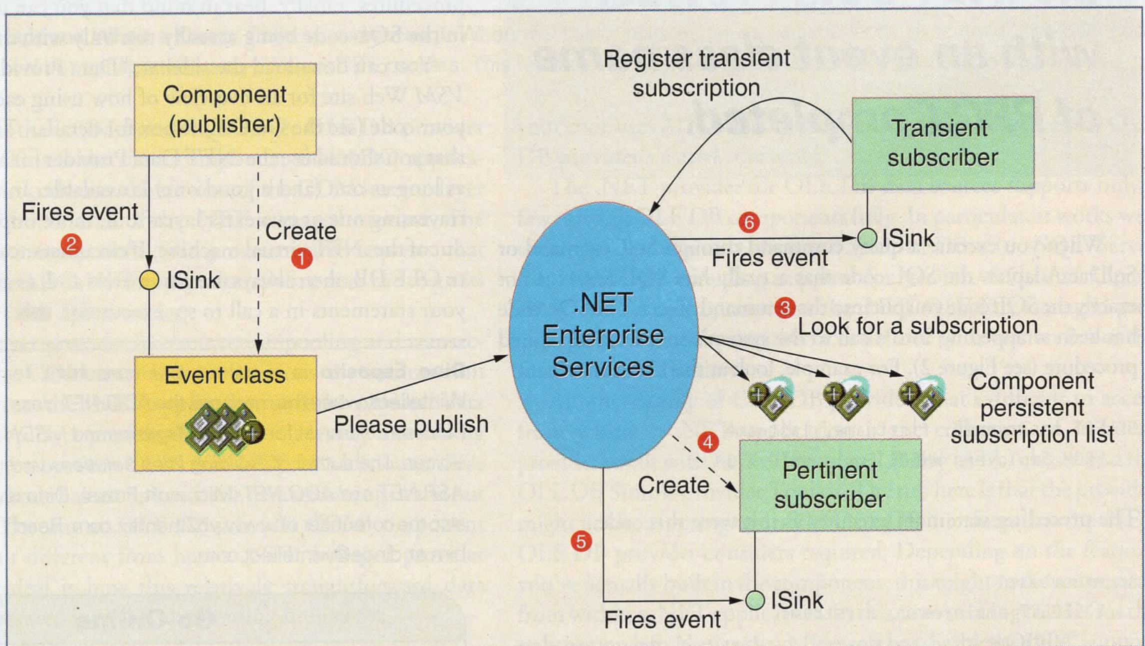

Figure 1 Use .NET as the Middleman for Events. Loosely Coupled Events (LCE) lets .NET mediate the publish/subscribe processes, simplifying code maintenance greatly and sometimes even improving performance, because it lets you manage how events fire. A publisher creates the event class ❶ and fires an event on it ❷. The .NET implementation of the event class goes through the list of subscribers for that event class ❸ and publishes the events to them (calling the appropriate sink method on the subscribers). .NET maintains a subscription list for every event class. Subscriptions can be transient (referring to existing objects) or persistent (the type of a class). For persistence subscriptions, .NET creates an object of every type stored ❹ and publishes to it ❺. After publishing to a persistent subscriber, the object is discarded and garbage-collected. For transient subscribers, .NET simply publishes the event to those objects ❻.

ods and set properties on the object. But what about the common situation where one or more clients needs to be notified about an object-side event? Most apps require some form of event subscription and publishing to handle this. You call the object publishing the event the *source* or *publisher*, and the party interested in the event a *sink* or *subscriber*.

Event notification takes place through publishers calling methods on subscribers. You can call publishing the event *firing* the event. .NET offers native support for events, relying on delegates (type-safe method references). .NET event support eases the task of managing events compared to COM connection points, and it bypasses the need to write code for managing subscriber lists.

However, .NET delegate-based events suffer from some drawbacks. A subscriber (or client adding a subscription) must duplicate the code for adding the subscription for every publisher object from which it wants to receive events. You can't subscribe to a type of event and have the event delivered to the subscriber, regardless of publisher. Subscribers can't filter events that are fired. For example, subscribers can't say, "Notify me about the event only if a certain condition is met." Subscribers must contact the publisher object in order to subscribe to it. This introduces coupling between clients and objects and between individual clients. Publishers and subscribers have coupled lifetimes—both must be running at the same time. Subscribers can't say to .NET, "If an object fires this particular event, create an instance of me and let me handle it." Finally, you must set up connections programmatically; you can't do it administratively.

Manage Events Loosely

You can overcome the drawbacks associated with delegate-based events in .NET by decoupling with LCE instead. You get LCE support from the System.EnterpriseServices namespace. .NET Enterprise Services actually form a thin wrapper around COM+ component services. LCE moves the logic for publishing and subscribing to events outside the scope of the components involved. Subscribers wanting to receive events register with .NET, then manage the subscribe/unsubscribe process through .NET, not the object. Similarly, publishers fire events at .NET rather than the subscribed clients.

LCE gives you a layer of indirection that decouples the subscriber and publisher in your system. No longer do your clients know anything about publishers' identities. And LCE differs from basic .NET delegate-based events in that it requires you to factor the event handling methods to an interface rather then mere public methods on the subscriber, as delegates require. This interface is called a *sink interface*.

A publisher object fires an event at .NET (to be delivered to the subscribers) using an event class, derived from ServicedComponent and decorated with the EventClass attribute. The event class must

Resources

- *COM and .NET Component Services* by Juval Löwy [O'Reilly & Associates, 2001, ISBN: 0596001037]
- Middle Tier, *Tap Into Transient Subscriptions*, by Jeff Prosise [*Visual C++ Developers Journal* March 2000]
- "Using COM+ Events": http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncomser/html/complus_events.asp

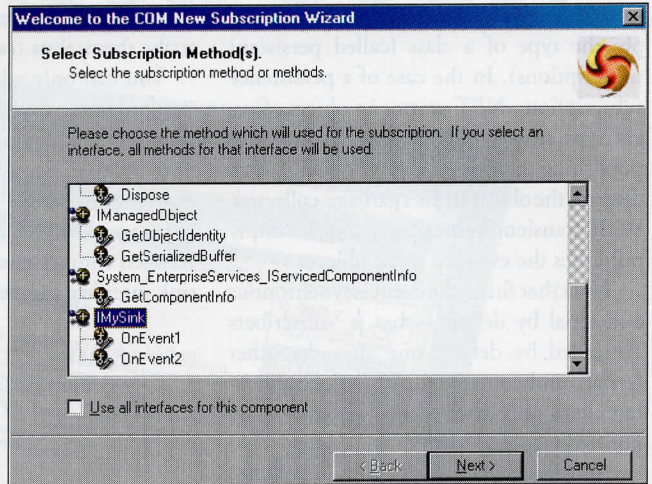

Figure 2 Simplify the Process With the Wizard. .NET's New Subscription Wizard simplifies implementing the publish/subscribe processes. It displays all the interfaces your component supports, including non-sink ones. It lets you subscribe to events published to all the sink interfaces your class supports, to a particular interface, or even to a particular method.

implement the sink interfaces used to publish the events. In reality, you never use this implementation because .NET synthesizes the actual implementation. You only use the event class to list the sink interfaces. This lets .NET synthesize their implementation.

Suppose a publishing object wants to fire an event at all the subscribers supporting the sink interface IMySink. These subscribers must both implement the interface and subscribe to the event class MyEventClass, defined in C# like this:

```
public interface IMySink
{
   void OnEvent1();
   void OnEvent2();
};
[EventClass]
public class MyEventClass :
   ServicedComponent,IMySink
{
   public void OnEvent1()
   {}
   public void OnEvent2()
   {}
};
```

To publish the event, the publisher first creates the event class, then fires the event at its interface's method (see Figure 1):

```
IMySink sink;
sink = new MyEventClass();
sink.OnEvent1();
```

The .NET implementation of the event class goes through the list of subscribers for that event class and publishes the events to them (calling the appropriate sink method on the subscribers). .NET maintains a list of subscriptions for every event class. Sub-

scriptions can be references to existing objects (called transient subscriptions) or simply the type of a class (called persistent subscriptions). In the case of a persistence subscription, .NET creates an object of every type stored and publishes to it. After publishing to a persistent subscriber, .NET discards the object and it's garbage-collected. With transient subscribers, .NET simply publishes the event to those objects.

Note that firing the event is synchronous and serial by default—that is, subscribers are called by default one after the other (serial), and control returns to the publishing client only after all the subscribers are notified (synchronous).

Another important distinction between LCE and delegate-based events is that if one of the subscribers throws an exception, only that subscriber is affected, and .NET continues to publish to the rest of the subscribers. With delegate-based events, if a subscriber throws an exception, the publisher must handle it or be terminated, and in any case won't be able to continue publishing.

Fine-Tune Delivery

However, .NET does provide you with some means of fine-tuning the delivery. You can minimize blocking time by configuring your event class to use multiple threads for publishing, using threads from the thread pool. You do this with the EventClass attribute's FireInParallel property:

```
[EventClass(FireInParallel = true)]
public class MyEventClass :
   ServicedComponent,IMySink
{
   public void OnEvent1(){}
   public void OnEvent2(){}
}
```

FireInParallel defaults to false. Parallel publishing is subject to pool limitations, so consider firing in parallel as an optimization technique only; avoid relying on it in your design. For example, don't count on all subscribers getting the event at the same time.

As I mentioned earlier, there are two types of subscribers. The first is an existing instance of a class supporting the sink interface. You can add that instance at run time to the list of subscribers of a particular event class. This is a *transient* subscription, which exists as long as the subscriber is running. It won't persist or survive a system reboot or crash.

When a particular instance of a class subscribes to an event class, only that instance receives events published. Other instances receive the events only if they subscribe themselves transiently.

You can only add a transient subscription programmatically, using .NET Enterprise Services Catalog interfaces and objects. You receive no administrative support through the .NET Component Services Explorer. Any .NET component (not just serviced components) that implements the sink interface can be a transient subscriber:

```
public class
   MyTransientSubscriber:
   IMySink
{...}
```

Use the second type of subscription—a *persistent* subscription—when you want .NET to create an object of a particular class type when an event is published, let it handle the event, then discard it. You can use only .NET classes derived from ServicedComponent as persistent subscribers:

```
public class MyPersistentSubscriber
      ServicedComponent, IMySink
{...}
```

Persistent subscriptions, as the name implies, persist in the .NET Catalog, and survive a system reboot or a crash.

Add Persistence

The process of adding a persistent subscription starts with registering the subscribing component with the Catalog (see Resources for more information on .NET Enterprise Services). Every component in the Component Services Explorer has a Subscription folder, which contains the persistent subscriptions the product administrator or developer has set up. Every subscription represents an event class (or a list of event classes). You instantiate the component to receive events whenever any publisher uses these event classes.

Next, expand the Subscription folder, right-click on it, and select New from the popup context menu. This invokes the New Subscription Wizard (see Figure 2). The wizard displays all the interfaces your component supports, including non-sink ones. .NET doesn't know whether they're sinks or not; only you do.

You can set up subscriptions at the interface or method level. Using the method level means that .NET delivers the event to your component only when publishers publish using that method. If you want to subscribe to another method, you must add a new subscription. Using the interface level means that any event targeting any method on that interface should be delivered to your component. These two options give you the ability to subscribe to only a subset of the events publishers can publish, or to all of them.

After you select interfaces and methods, the wizard lists all installed event classes supporting the interfaces you've selected. You can choose a particular event class or all of them. The last step in the wizard lets you name the subscription and enable it.

You can enable or disable a subscription easily after creating it. Highlight it in the Subscriptions folder, display its Properties page, select the Options tab, and enable or disable the subscription.

Delegates are great, but you need LCE in your toolkit too. Once you master persistent subscriptions, you can move on to transient ones, discussed in the online sidebar, "Add a Transient Subscription" (see the Go Online box for details). Either way, you won't really start exploiting publishing and subscribing with .NET until you master LCE. VSM

Juval Löwy is a software architect, conference speaker, and principal of IDesign, a consulting and training company focused on .NET design and migration. This article is based on his book, *COM and .NET Component Services* (O'Reilly & Associates). Juval is an officer of the .NET California Bay Area User Group. Reach him at www.idesign.net.

Go Online

Use these Locator+ codes at www.visualstudiomagazine.com to go directly to these related resources.

VS0206 Download all the code for this issue of *VSM*.

VS0206BB Download the code for this article separately. This article's code includes the .NET LCE-Persistent class library with an event class and persistent subscriber, plus a Windows Forms test client to publish the event, making .NET instantiate the subscriber and deliver the event to it. The file also includes a sidebar and listings showing how to code transient subscriptions, a transient subscription helper class, and a test transient subscriber and publisher.

VS0206BB_T Read this article online.