

51

Enterprise Services

WHAT'S IN THIS CHAPTER?

- Features of Enterprise Services
- Using Enterprise Services
- Creating a serviced component
- Deploying COM+ applications
- Using transactions with COM+
- Creating a WCF façade to Enterprise Services
- Using Enterprise Services from a WCF client

Enterprise Services is the name of the Microsoft application server technology that offers services for distributed solutions. Enterprise Services is based on the COM+ technology that has already been in use for many years. However, instead of wrapping .NET objects as COM objects to use these services, .NET offers extensions for .NET components to take direct advantage of these services. With .NET you get easy access to COM+ services for .NET components.

Enterprise Services also has a great integration story with Windows Communication Foundation (WCF). You can use a tool to automatically create a WCF service front-end to a serviced component, and you can invoke a WCF service from a COM+ client.



This chapter uses the sample database Northwind, which you can download from the Microsoft downloads page at www.microsoft.com/downloads.

USING ENTERPRISE SERVICES

The complexity of Enterprise Services and the different configuration options (many of them are not needed if all the components of the solution are developed with .NET) can be more easily understood if you know the history of Enterprise Services. This section starts with that history.

You then get an overview of the different services offered by the technology, so you know what features could be useful for your application.

This section describes the history of Enterprise Services, contexts that are the foundation of the functionality, and the key features such as automatic transactions, object pooling, role-based security, queued components, and loosely coupled events

History

Enterprise Services can be traced back to Microsoft Transaction Server (MTS), which was released as an option pack for Windows NT 4.0. MTS extended COM by offering services such as transactions for COM objects. The services could be used by configuring metadata: the configuration of the component defined whether or not a transaction was required. With MTS it was no longer necessary to deal with transactions programmatically. However, MTS had a big disadvantage. COM was not designed to be extensible, so MTS made extensions by overwriting the COM component registry configuration to direct the instantiation of the component to MTS, and some special MTS API calls have been required to instantiate COM objects within MTS. This problem was solved with Windows 2000.

One of the most important features of Windows 2000 was the integration of MTS and COM in a new technology with the name COM+. In Windows 2000, COM+ base services are aware of the context that is needed by COM+ services (previously MTS services), so the special MTS API calls are no longer needed. With COM+ services some new service functionality is offered in addition to distributed transactions.

Windows 2000 includes COM+ 1.0. COM+ 1.5 has been available since Windows XP and Windows Server 2003. COM+ 1.5 adds more features to increase scalability and availability, including application pooling and recycling, and configurable isolation levels.

.NET Enterprise Services allows you to use COM+ services from within .NET components. Support is offered for Windows 2000 and later. When .NET components are run within COM+ applications, no COM callable wrapper is used (see Chapter 26, “Interop”); the application runs as a .NET component instead. When you install the .NET runtime on an operating system, some runtime extensions are added to COM+ services. If two .NET components are installed with Enterprise Services, and component A is using component B, COM marshaling is not used; instead, the .NET components can invoke each other directly.

Where to Use Enterprise Services

Business applications can be logically separated into presentation, business, and data service layers. The *presentation service layer* is responsible for user interaction. Here, the user can interact with the application to enter and view data. Technologies used with this layer are Windows Forms, WPF, and ASP.NET. The *business service layer* consists of business rules and data rules. The *data service layer* interacts with persistent storage. Here, you can use components that make use of ADO.NET. Enterprise Services fits both to the business service layer and to the data service layer.

Figure 51-1 shows two typical application scenarios. Enterprise Services can be used directly from a rich client using Windows Forms or WPF or from a web application that is running ASP.NET.

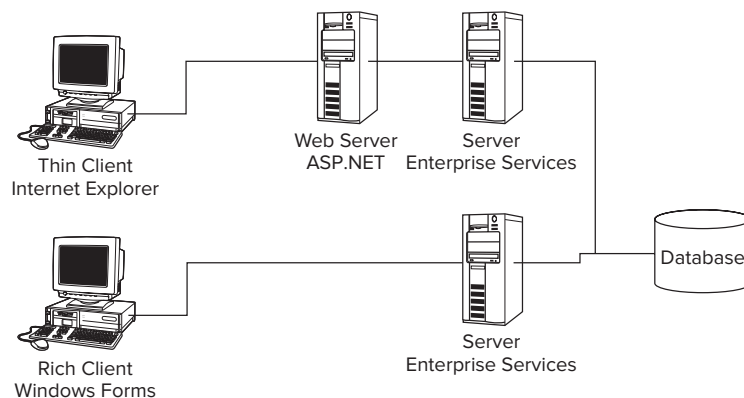


FIGURE 51-1

Enterprise Services is also a scalable technology. Using *component load balancing* makes it possible to distribute the load of the clients across different systems.

You can also use Enterprise Services on the client system, because this technology is included in Windows 7 and Windows Vista.

Key Features

Now let's get into the advantages of using Enterprise Services by looking at its key features, such as automatic transactions, role-based security, queued components, and loosely coupled events.

Contexts

The base functionality behind the services offered by Enterprise Services is the context that is the foundation of all the features of Enterprise Services. The context makes it possible to intercept a method call, and some service functionality can be carried out before the expected method call is invoked. For example, a transactional or a synchronization scope can be created before the method implemented by the component is invoked. Figure 51-2 shows object A and object B running in two different contexts, X and Y. Between the contexts the call is intercepted by a proxy. The proxy can use services offered by Enterprise Services that are explained with the following features.

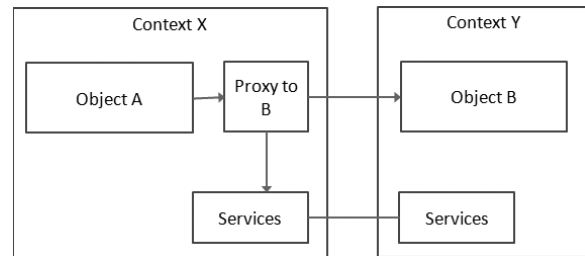


FIGURE 51-2

With the contexts here, a COM component and a .NET component can participate in the same transaction. All this is done with the help of the base class *ServicedComponent* that itself derives from *MarshalByRefObject* to integrate .NET and COM+ contexts.

Automatic Transactions

The most commonly used feature of Enterprise Services is *automatic transactions*. With automatic transactions, it is not necessary to start and commit a transaction in the code; an attribute can be applied to a class instead. By using the `[Transaction]` attribute with the options `Required`, `Supported`, `RequiresNew`, and `NotSupported`, you can mark a class with the requirements it has for transactions. If you mark the class with the option `Required`, a transaction is created automatically when a method starts and is committed to or aborted when the root component of the transaction is finished.

Such a declarative way to program is of particular advantage when a complex object model is developed. Here, automatic transactions have a big advantage over programming transactions manually. Assume that you have a *Person* object with multiple *Address* and *Document* objects that are associated with the *Person*, and you want to store the *Person* object together with all associated objects in a single transaction. Doing transactions programmatically would mean passing a transaction object to all the related objects so that they can participate in the same transaction. Using transactions declaratively means there is no need to pass the transaction object, because this happens behind the scenes by using the context.

Distributed Transactions

Enterprise Services not only offers automatic transactions, but the transactions can also be distributed across multiple databases. Enterprise Services transactions are enlisted with the *Distributed Transaction Coordinator (DTC)*. The DTC supports databases that make use of the XA protocol, which is a two-phase commit protocol, and is supported by SQL Server and Oracle. A single transaction can span writing data to both a SQL Server and an Oracle database.

Distributed transactions are not only useful with databases, but a single transaction can also span writing data to a database and writing data to a message queue. If one of these two actions fails, a rollback is done with the other action. You can read more about message queuing in Chapter 46, "Message Queuing."



Enterprise Services supports promotable transactions. If SQL Server is used and just a single connection is active within one transaction, a local transaction is created. If another transactional resource is active within the same transaction, the transaction is promoted to a DTC transaction.

Later in this chapter, you see how to create a component that requires transactions.

Object Pooling

Pooling is another feature offered by Enterprise Services. These services use a pool of threads to answer requests from clients. Object pooling can be used for objects with a long initialization time. With object pooling, objects are created in advance so that clients don't have to wait until the object is initialized.

Role-Based Security

Using *role-based security* allows you to define roles declaratively and define what methods or components can be used from what roles. The system administrator assigns users or user groups to these roles. In the program there is no need to deal with access control lists; instead, roles that are simple strings can be used.

Queued Components

Queued components is an abstraction layer to message queuing. Instead of sending messages to a message queue, the client can invoke methods with a recorder that offers the same methods as a .NET class configured in Enterprise Services. The recorder in turn creates messages that are transferred via a message queue to the server application.

Queued components and message queuing are useful if the client application is running in a disconnected environment (for example, on a laptop that does not always have a connection to the server), or if the request that is sent to the server should be cached before it is forwarded to a different server (for example, to a server of a partner company).

Loosely Coupled Events

Chapter 8, “Delegates, Lambdas, and Events,” explains the event model of .NET. Chapter 26 describes how to use events in a COM environment. With both of these event mechanisms, the client and the server have a tight connection. This is different with *loosely coupled events (LCE)*. With LCE the COM+ facility is inserted between client and server (see Figure 51-3). The publisher registers the events it will offer with COM+ by defining an event class. Instead of sending the events directly to the client, the publisher sends events to the event class that is registered with the LCE service. The LCE service forwards the events to the subscriber, which is the client application that registered a subscription for the event.

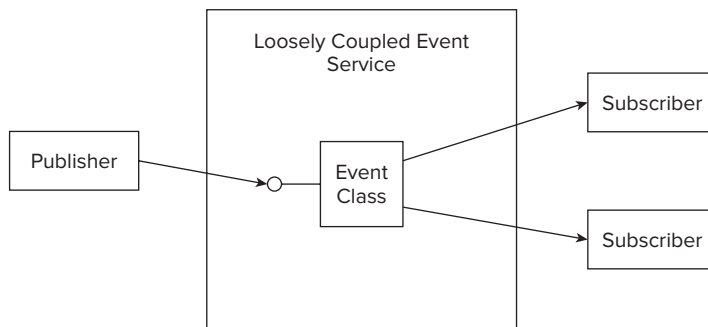


FIGURE 51-3

CREATING A SIMPLE COM+ APPLICATION

To create a .NET class that can be configured with Enterprise Services, you have to reference the assembly `System.EnterpriseServices` and add the namespace `System.EnterpriseServices` to the using declarations. The most important class to use is `ServiceComponent`.

The first example shows the basic requirements to create a serviced component. You start by creating a C# library application. All COM+ applications must be written as library applications regardless of whether they will run in their own process or in the process of the client. Name the library `SimpleServer`. Reference the assembly `System.EnterpriseServices` and add the declaration using `System.EnterpriseServices`; to the `AssemblyInfo.cs` and `Class1.cs` files.

With the project you also need to apply a strong name to the library. For some Enterprise Services features it is also necessary to install the assembly in the GAC (global assembly cache). Strong names and the global assembly cache are discussed in Chapter 18, “Assemblies.”

The ServiceComponent Class

Every serviced component class must derive from the base class `ServiceComponent`. `ServiceComponent` itself derives from the class `ContextBoundObject`, so an instance is bound to a .NET Remoting context.

The class `ServiceComponent` has some protected methods that can be overridden, as shown in the following table.

PROTECTED METHOD	DESCRIPTION
<code>Activate()</code> <code>Deactivate()</code>	The <code>Activate()</code> and <code>Deactivate()</code> methods are called if the object is configured to use object pooling. When the object is taken from the pool, the <code>Activate()</code> method is called. Before the object is returned to the pool, <code>Deactivate()</code> is called.
<code>CanBePooled()</code>	This is another method for object pooling. If the object is in an inconsistent state, you can return <code>false</code> in your overridden implementation of <code>CanBePooled()</code> . This way the object is not put back into the pool, but destroyed instead. A new object is created for the pool.
<code>Construct()</code>	This method is called at instantiation time, where a construction string can be passed to the object. The construction string can be modified by the system administrator. Later in this chapter, you use the construction string to define the database connection string.

Assembly Attributes

Some Enterprise Services attributes are also needed. The attribute `ApplicationName` defines the name of the application as it will be seen in the Component Services Explorer. The value of the `Description` attribute shows up as a description within the application configuration tool.

`ApplicationActivation` allows you to define whether the application should be configured as a library application or a server application, using the options `ActivationOption.Library` or `ActivationOption.Server`. With a library application, the application is loaded inside the process of the client. In that case the client might be the ASP.NET runtime. With a server application, a process for the application is started. The name of the process is `dllhost.exe`. With the attribute `ApplicationAccessControl`, you can turn off security so that every user is allowed to use the component.

Rename the file `Class1.cs` to `SimpleComponent.cs` and add these attributes outside the namespace declaration:

```
[assembly: ApplicationName("Wrox EnterpriseDemo")]
[assembly: Description("Wrox Sample Application for Professional C#")]
[assembly: ApplicationActivation(ActivationOption.Server)]
[assembly: ApplicationAccessControl(false)]
```

The following table lists the most important assembly attributes that can be defined with Enterprise Services applications.

ATTRIBUTE	DESCRIPTION
ApplicationName	The attribute <code>ApplicationName</code> defines the name for the COM+ application that shows up in the Component Services Explorer after the component is configured.
ApplicationActivation	The attribute <code>ApplicationActivation</code> defines if the application should run as a library within the client application or if a separate process should be started. The options to configure are defined with the enumeration <code>ActivationOption</code> . <code>ActivationOption.Library</code> defines to run the application inside the process of the client; <code>ActivationOption.Server</code> starts its own process, <code>dllhost.exe</code> .
ApplicationAccessControl	The attribute <code>ApplicationAccessControl</code> defines the security configuration of the application. Using a Boolean value you can enable or disable access control. With the <code>Authentication</code> property you can set privacy levels — whether the client should be authenticated with every method call or just with the connection, and whether the data sent should be encrypted.

Creating the Component

In the `SimpleComponent.cs` file, you can create your serviced component class. With serviced components, it is best to define interfaces that are used as the contract between the client and the component. This is not a strict requirement, but some of the Enterprise Services features (such as setting role-based security on a method or interface level) require interfaces. Create the interface `IGreeting` with the method `Welcome()`. The attribute `ComVisible` is required for serviced component classes and interfaces that can be accessed from Enterprise Services features:



Available for download on Wrox.com

```
using System.Runtime.InteropServices;

namespace Wrox.ProCSharp.EnterpriseServices
{
    [ComVisible(true)]
    public interface IGreeting
    {
        string Welcome(string name);
    }
}
```

code snippet SimpleServer/IGreeting.cs

The class `SimpleComponent` derives from the base class `ServicedComponent` and implements the interface `IGreeting`. The class `ServicedComponent` acts as a base class of all serviced component classes, and offers some methods for the activation and construction phases. Applying the attribute `EventTrackingEnabled` to this class makes it possible to monitor the objects with the Component Services Explorer. By default, monitoring is disabled because using this feature reduces performance. The `Description` attribute only specifies text that shows up in the Explorer:



Available for download on Wrox.com

```
using System.EnterpriseServices;
using System.Runtime.InteropServices;

namespace Wrox.ProCSharp.EnterpriseServices
{
    [EventTrackingEnabled(true)]
    [ComVisible(true)]
    [Description("Simple Serviced Component Sample")]
    public class SimpleComponent: ServicedComponent, IGreeting
    {
    }
}
```

```
{
    public SimpleComponent()
    {
    }
}
```

code snippet SimpleServer/SimpleComponent.cs

The method `Welcome()` returns only "Hello, " with the name that is passed to the argument. So that you can see some visible result in the Component Services Explorer while the component is running, `Thread.Sleep()` simulates some processing time:

```
        public string Welcome(string name)
        {
            // simulate some processing time
            System.Threading.Thread.Sleep(1000);
            return "Hello, " + name;
        }
    }
}
```

Other than applying some attributes and deriving the class from `ServiceComponent`, there's nothing special to do with classes that should use Enterprises Services features. All that is left to do is build and deploy a client application.

In the first sample component, the attribute `[EventTrackingEnabled]` was set. Some more commonly used attributes that influence the configuration of serviced components are described in the following table.

ATTRIBUTE CLASS	DESCRIPTION
<code>EventTrackingEnabled</code>	Setting the attribute <code>EventTrackingEnabled</code> allows monitoring the component with the Component Services Explorer. Setting this attribute to <code>true</code> has some additional overhead associated; that's why, by default, event tracking is turned off.
<code>JustInTimeActivation</code>	With this attribute, the component can be configured to not activate when the caller instantiates the class, but instead when the first method is invoked. Also, with this attribute the component can deactivate itself.
<code>ObjectPooling</code>	If the initialization time of a component is long compared to the time of a method call, an object pool can be configured with the attribute <code>ObjectPooling</code> . With this attribute, minimum and maximum values can be defined that influence the number of objects in the pool.
<code>Transaction</code>	The attribute <code>Transaction</code> defines transactional characteristics of the component. Here, the component defines whether a transaction is required, supported, or not supported.

DEPLOYMENT

Assemblies with serviced components must be configured with COM+. This configuration can be done automatically or by registering the assembly manually.

Automatic Deployment

If a .NET client application that uses the serviced component is started, the COM+ application is configured automatically. This is true for all classes that are derived from the class `ServiceComponent`. Application and class attributes such as `EventTrackingEnabled` define the characteristics of the configuration.

Automatic deployment has an important drawback. For automatic deployment to work, the client application needs administrative rights. If the client application that invokes the serviced component is ASP.NET, the ASP.NET runtime usually doesn't have administrative rights. With this drawback, automatic deployment is useful only during development time. However, during development, automatic deployment is an extremely advantageous feature because it is not necessary to do manual deployment after every build.

Manual Deployment

You can deploy the assembly manually with the command-line utility .NET Services installation tool `regsvcs.exe`. Starting the command

```
regsvcs SimpleServer.dll
```

registers the assembly `SimpleServer` as a COM+ application and configures the included components according to their attributes; it also creates a type library that can be used by COM clients accessing the .NET component.

After you've configured the assembly, you can start the Component Services Explorer by selecting Administrative Tools ⇨ Component Services. In the left tree view of this application, you can select Component Services ⇨ Computers ⇨ My ⇨ Computer ⇨ COM+ Applications to verify that the application was configured.



On Windows Vista you have to start the MMC and add the Component Services snap in to see the Component Services Explorer.

Creating an Installer Package

With the Component Services Explorer, you can create Windows installer packages for server or client systems. An installer package for the server includes the assemblies and configuration settings to install the application on a different server. If the serviced component is invoked from applications running on different systems, a proxy must be installed on the client system. The installer package for the client includes assemblies and configuration for proxies.

To create an installer package, you can start the Component Services Explorer, select the COM+ application, select the menu options Action ⇨ Export, and click the Next button in the first dialog. The dialog shown in Figure 51-4 opens. Within this box you can export either a Server application or an Application proxy. With the option Server application you can also configure to Export user identities with roles. This option should be selected only if the target system is in the same domain as the system where the package is created, because the configured user identities are put into the installer package. With Application proxy, an installer package for the client system is created.

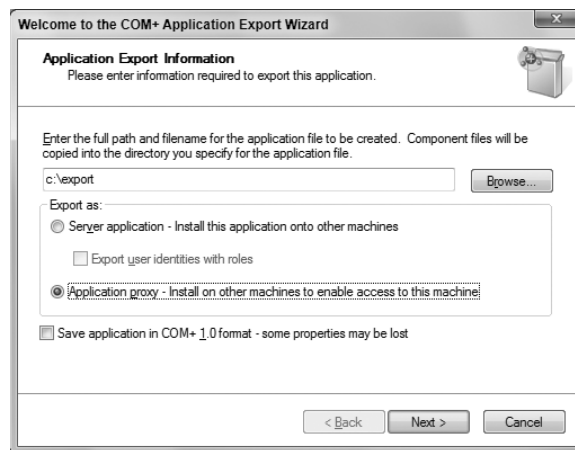


FIGURE 51-4



The option to create an application proxy is not available if the application is configured as a library application.

To install the proxy, you have to start `setup.exe` from the installer package. Be aware that an application proxy cannot be installed on the same system where the application is installed. After installation of the application proxy, you can see an entry in Component Services Explorer that represents the application proxy. With the application proxy the only option that can be configured is the name of the server in the Activation tab, as discussed in the next section.

COMPONENT SERVICES EXPLORER

After a successful configuration, you can see Wrox EnterpriseDemo as an application name in the tree view of the Component Services Explorer. This name was set by the attribute [ApplicationName]. Selecting **Action** ⇨ **Properties** opens the dialog shown in Figure 51-5. Both the name and the description have been configured by using attributes. When you select the **Activation** tab, you can see that the application is configured as a server application because it has been defined with the [ApplicationActivation] attribute, and selecting the **Security** tab shows that the Enforce access checks for this application option is not selected because the attribute [ApplicationAccessControl] was set to false.

The following is a list of more options that can be set with this application:

- **Security** — With security configuration, you can enable or disable access checks. If security is enabled, you can set access checks to the application level, the component, the interface, and to the method level. It is also possible to encrypt messages that are sent across the network using packet privacy as an authentication level for calls. Of course, this also increases the overhead.
- **Identity** — With server applications, you can use the Identity tab to configure the user account that will be used for the process that hosts the application. By default, this is the interactive user. This setting is very useful while debugging the application but cannot be used on a production system if the application is running on a server, because there might not be anybody logged on. Before installing the application on the production system you should test the application by using a specific user for the application.
- **Activation** — The Activation tab allows you to configure the application either as a library or as a server application. Two new options with COM+ 1.5 are the option to run the application as a Windows Service and to use SOAP to access the application. Windows Services are discussed in Chapter 25, “Windows Services.” Selecting the SOAP option uses .NET Remoting configured within Internet Information Server to access the component. Instead of using .NET Remoting, later in this chapter the component will be accessed using WCF. WCF is discussed in Chapter 43, “Windows Communication Foundation.”

With an application proxy, the option “Remote server name” is the only option that can be configured. This option sets the name of the server. By default, the DCOM protocol is used as the network protocol. However, if SOAP is selected in the server configuration, the communication happens through .NET Remoting.

- **Queuing** — The Queuing configuration is required for service components that make use of message queuing.
- **Advanced** — On the Advanced tab, you can specify whether the application should be shut down after a certain period of client inactivity. You can also specify whether to lock a certain configuration so that no one can change it accidentally.
- **Dump** — If the application crashes, you can specify the directory where the dumps should be stored. This is useful for components developed with C++.
- **Pooling and Recycling** — Pooling and recycling is a new option with COM+ 1.5. With this option, you can configure whether the application should be restarted (recycled) depending on application lifetime, memory needs, number of calls, and so on.

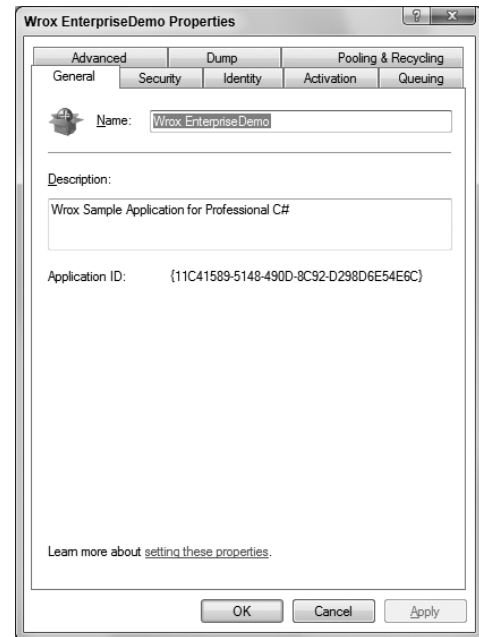


FIGURE 51-5

With the Component Services Explorer, you can also view and configure the component itself. When opening child elements of the application, you can view the component `Wrox.ProCSharp.EnterpriseServices.SimpleComponent`. Selecting Action Properties opens the dialog shown in Figure 51-6.

Using this dialog, you can configure these options:

- **Transactions** — On the Transactions tab, you can specify whether the component requires transactions. You use this feature in the next example.
- **Security** — If security is enabled for the application, you can define what roles are allowed to use the component with this configuration.
- **Activation** — The Activation configuration enables you to set object pooling and to assign a construction string.
- **Concurrency** — If the component is not thread-safe, concurrency can be set to Required or Requires New. This way the COM+ runtime allows only one thread at a time to access the component.

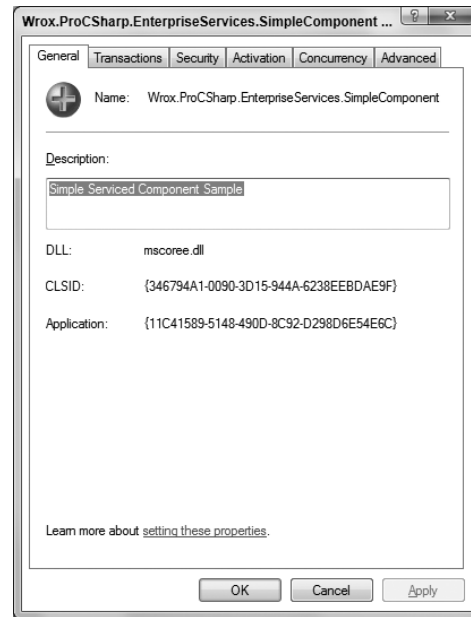


FIGURE 51-6

CLIENT APPLICATION

After building the serviced component library, you can create a client application. This can be as simple as a C# console application. After you've created the project for the client, you have to reference both the assembly from the serviced component, `SimpleServer`, and the assembly `System.EnterpriseServices`. Then you can write the code to instantiate a new `SimpleComponent` instance and invoke the method `Welcome()`. In the following code, the `Welcome()` method is called 10 times. The `using` statement helps release the resources allocated with the instance before the garbage collector takes action. With the `using` statement, the `Dispose()` method of the serviced component is called when the scope of the `using` statement ends:

using System;

```
namespace Wrox.ProCSharp.EnterpriseServices
{
    class Program
    {
        static void Main()
        {
            using (SimpleComponent obj = new SimpleComponent())
            {
                for (int i = 0; i < 10; i++)
                {
                    Console.WriteLine(obj.Welcome("Stephanie"));
                }
            }
        }
    }
}
```



code snippet ClientApplication/Program.cs

If you start the client application before configuring the server, the server will be configured automatically. The automatic configuration of the server is done with the values that you've specified using attributes. For a test you can unregister the serviced component and start the client again. If the serviced component is configured during the start of the client application, the startup needs more time. Remember that

this feature is useful only during development time. Administrative rights are also needed for automatic deployment. If you are starting the application from within Visual Studio, you should start Visual Studio with administrative rights.

While the application is running, you can monitor the serviced component with the Component Services Explorer. By selecting Components in the tree view and choosing View ⇄ Detail, you can view the number of instantiated objects if the attribute [EventTrackingEnabled] is set.

As you've seen, creating serviced components is just a matter of deriving the class from the base class `ServicedComponent` and setting some attributes to configure the application. Next, you see how transactions can be used with serviced components.

TRANSACTIONS

Automatic transactions are the most frequently used feature of Enterprise Services. Using Enterprise Services, you can mark the components as requiring a transaction, and the transaction is then created from the COM+ runtime. All transaction-aware objects inside the component, such as ADO.NET connections, run inside the transaction.



You can read more about the concepts of transactions in Chapter 23, "System.Transactions."

Transaction Attributes

Serviced components can be marked with the [Transaction] attribute to define if and how transactions are required for the component.

Figure 51-7 shows multiple components with different transactional configurations. The client invokes component A. Because component A is configured with `Transaction Required` and no transaction existed previously, the new transaction, 1, is created. Component A invokes component B, which in turn invokes component C. Because component B is configured with `Transaction Supported`, and the configuration of component C is set to `Transaction Required`, all three components (A, B, and C) use the same transaction context. If component B were configured with the transaction setting `NotSupported`, component C would get a new transaction. Component D is configured with the setting `New Transaction Required`, so a new transaction is created when it is called by component A.

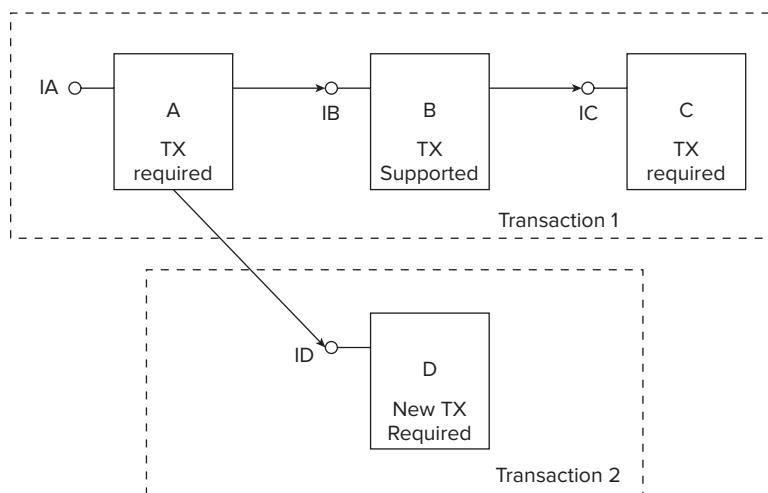


FIGURE 51-7

The following table lists the different values that you can set with the `TransactionOption` enumeration:

TRANSACTIONOPTION VALUE	DESCRIPTION
Required	Setting the [Transaction] attribute to <code>TransactionOption.Required</code> means that the component runs inside a transaction. If a transaction has been created already, the component will run in the same transaction. If no transaction exists, a transaction is created.
RequiresNew	<code>TransactionOption.RequiresNew</code> always results in a newly created transaction. The component never participates in the same transaction as the caller.
Supported	With <code>TransactionOption.Supported</code> , the component doesn't need transactions itself. However, the transaction spans the caller and the called component, if these components require transactions.
NotSupported	The option <code>TransactionOption.NotSupported</code> means that the component never runs in a transaction, regardless of whether the caller has a transaction.
Disabled	<code>TransactionOption.Disabled</code> means that a possible transaction of the current context is ignored.

Transaction Results

A transaction can be influenced by setting the *consistent* and the *done* bit of the context. If the consistent bit is set to `true`, the component is happy with the outcome of the transaction. The transaction can be committed if all components participating with the transaction are similarly successful. If the consistent bit is set to `false`, the component is not happy with the outcome of the transaction, and the transaction is aborted when the root object that started the transaction is finished. If the done bit is set, the object can be deactivated after the method call ends. A new instance is created with the next method call.

The consistent and done bits can be set using four methods of the `ContextUtil` class with the results that you can see in the following table.

CONTEXTUTIL METHOD	CONSISTENT BIT	DONE BIT
<code>SetComplete</code>	<code>True</code>	<code>true</code>
<code>SetAbort</code>	<code>False</code>	<code>true</code>
<code>EnableCommit</code>	<code>True</code>	<code>false</code>
<code>DisableCommit</code>	<code>False</code>	<code>false</code>

With .NET it is also possible to set the consistent and done bit by applying the attribute `[AutoComplete]` to the method instead of calling the `ContextUtil` methods. With this attribute the method `ContextUtil.SetComplete()` is called automatically if the method is successful. If the method fails and an exception is thrown, with `[AutoComplete]` the method `ContextUtil.SetAbort()` is called.

SAMPLE APPLICATION

This sample application simulates a simplified scenario that writes new orders to the Northwind sample database. As shown in Figure 51-8, multiple components are used with the COM+ application. The class `OrderControl` is called from the client application to create new orders. `OrderControl` uses the `OrderData` component. `OrderData` has the responsibility of creating a new entry in the `Order` table of the Northwind database. The `OrderData` component uses the `OrderLineData` component to write `OrderDetail` entries to the database. Both `OrderData` and `OrderLineData` must participate in the same transaction.

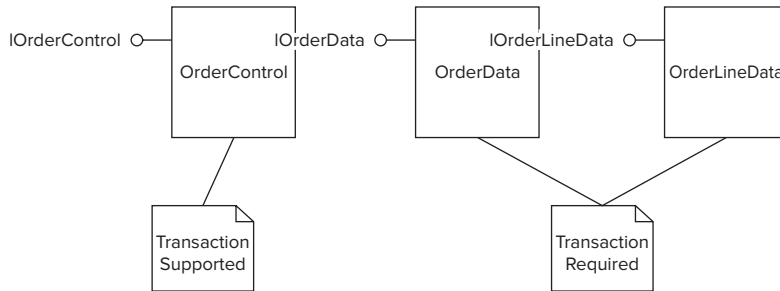


FIGURE 51-8

Start by creating a C# Component library with the name `NorthwindComponents`. Sign the assembly with a keyfile, and define the Enterprise Services application attributes as shown in the following code:



```
[assembly: ApplicationName("Wrox.NorthwindDemo")]
[assembly: ApplicationActivation(ActivationOption.Server)]
[assembly: ApplicationAccessControl(false)]
```

code snippet NorthwindComponents/AssemblyInfo.cs

Entity Classes

Next add the entity classes `Order` and `OrderLine` that represent the columns in the Northwind database tables `Order` and `Order Details`. Entity classes are data holders representing the data that is important for the application domain — in that case for doing orders. The class `Order` has a static method `Create()` that creates and returns a new instance of the class `Order`, and initializes this instance with the arguments passed to this method. Also, the class `Order` has some read-only properties `OrderId`, `CustomerId`, `OrderDate`, `ShipAddress`, `ShipCity`, and `ShipCountry`. The value of the `OrderId` property is not known at creation time of the class `Order`, but because the `Order` table in the Northwind database has an auto-increment attribute, the value is known after the order is written to the database. The method `SetOrderId()` is used to set the corresponding ID after the order has been written to the database. Because this method is called by a class inside the same assembly, the access level of this method is set to `internal`. The method `AddOrderLine()` adds order details to the order:



```
using System;
using System.Collections.Generic;

namespace Wrox.ProCSharp.EnterpriseServices
{
    [Serializable]
    public class Order
    {
        public static Order Create(string customerId, DateTime orderDate,
                                   string shipAddress, string shipCity, string shipCountry)
        {
            return new Order
            {
                CustomerId = customerId,
                OrderDate = orderDate,
                ShipAddress = shipAddress,
                ShipCity = shipCity,
                ShipCountry = shipCountry
            };
        }

        public Order()
        {
        }
    }
}
```

```

        internal void SetOrderId(int orderId)
        {
            this.OrderId = orderId;
        }

        public void AddOrderLine(OrderLine orderLine)
        {
            orderLines.Add(orderLine);
        }

        private readonly List<OrderLine> orderLines = new List<OrderLine>();

        public int OrderId { get; private set; }
        public string CustomerId { get; private set; }
        public DateTime OrderDate { get; private set; }
        public string ShipAddress { get; private set; }
        public string ShipCity { get; private set; }
        public string ShipCountry { get; private set; }

        public OrderLine[] OrderLines
        {
            get
            {
                OrderLine[] ol = new OrderLine[orderLines.Count];
                orderLines.CopyTo(ol);
                return ol;
            }
        }
    }
}

```

code snippet NorthwindComponents/Order.cs

The second entity class is OrderLine. OrderLine has a static Create() method similar to the one of the Order class. Other than that, the class only has some properties ProductId, UnitPrice, and Quantity:



Available for
download on
Wrox.com

```

using System;

namespace Wrox.ProCSharp.EnterpriseServices
{
    [Serializable]
    public class OrderLine
    {
        public static OrderLine Create(int productId, float unitPrice, int quantity)
        {
            return new OrderLine
            {
                ProductId = productId,
                UnitPrice = unitPrice,
                Quantity = quantity
            };
        }

        public OrderLine()
        {
        }

        public int ProductId { get; set; }
        public float UnitPrice { get; set; }
        public int Quantity { get; set; }
    }
}

```

code snippet NorthwindComponents/OrderLine.cs

The OrderControl Component

The class `OrderControl` represents a simple business services component. In this example, just one method, `NewOrder()`, is defined in the interface `IOrderControl`. The implementation of `NewOrder()` does nothing more than instantiate a new instance of the data services component `OrderData` and call the method `Insert()` to write an `Order` object to the database. In a more complex scenario, this method could be extended to write a log entry to a database or to invoke a queued component to send the `Order` object to a message queue:



```
using System;
using System.EnterpriseServices;
using System.Runtime.InteropServices;

namespace Wrox.ProCSharp.EnterpriseServices
{
    [ComVisible(true)]
    public interface IOrderControl
    {
        void NewOrder(Order order);
    }

    [Transaction(TransactionOption.Supported)]
    [EventTrackingEnabled(true)]
    [ComVisible(true)]
    public class OrderControl: ServicedComponent, IOrderControl
    {
        [AutoComplete()]
        public void NewOrder(Order order)
        {
            using (OrderData data = new OrderData())
            {
                data.Insert(order);
            }
        }
    }
}
```

code snippet NorthwindComponents/OrderControl.cs

The OrderData Component

The `OrderData` class is responsible for writing the values of `Order` objects to the database. The interface `IOrderUpdate` defines the `Insert()` method. You can extend this interface to also support an `Update()` method where an existing entry in the database is updated:



```
using System;
using System.Data.SqlClient;
using System.EnterpriseServices;
using System.Runtime.InteropServices;

namespace Wrox.ProCSharp.EnterpriseServices
{
    [ComVisible(true)]
    public interface IOrderUpdate
    {
        void Insert(Order order);
    }
}
```

code snippet NorthwindComponents/OrderData.cs

The class `OrderData` has the attribute `[Transaction]` with the value `TransactionOption.Required` applied. This means that the component will run in a transaction in any case. Either a transaction is

created by the caller and `OrderData` uses the same transaction, or a new transaction is created. Here a new transaction is created because the calling component `OrderControl` doesn't have a transaction.

With serviced components, you can use only default constructors. However, you can use the Component Services Explorer to configure a construction string that is sent to a component (see Figure 51-9). Selecting the Activation tab of the component configuration enables you to change the constructor string. The option "Enable object construction" is turned on when the attribute `ConstructionEnabled` is set, as it is with the class `OrderData`. The `Default` property of the `ConstructionEnabled` attribute defines the default connection string shown in the Activation settings after registration of the assembly. Setting this attribute also requires you to overload the method `Construct()` from the base class `ServicedComponent`. This method is called by the COM+ runtime at object instantiation, and the construction string is passed as an argument. The construction string is set to the variable `connectionString`, which is used later to connect to the database:

```
[Transaction(TransactionOption.Required)]
[EventTrackingEnabled(true)]
[ConstructionEnabled
    (true, Default="server=(local);database=northwind;trusted_connection=true")]
[ComVisible(true)]
public class OrderData: ServicedComponent, IOrderUpdate
{
    private string connectionString;

    protected override void Construct(string s)
    {
        connectionString = s;
    }
}
```

The method `Insert()` is at the heart of the component. Here, you use ADO.NET to write the `Order` object to the database. (ADO.NET is discussed in more detail in Chapter 30, "Core ADO.NET.") In this example, you create a `SqlConnection` object where the connection string that was set with the `Construct()` method is used to initialize the object.

The attribute `AutoComplete` is applied to the method to use automatic transaction handling as discussed earlier:

```
[AutoComplete()]
public void Insert(Order order)
{
    var connection = new SqlConnection(
        connectionString);
```

The method `connection.CreateCommand()` creates a `SqlCommand` object where the `CommandText` property is set to a SQL `INSERT` statement to add a new record to the `Orders` table. The method `ExecuteNonQuery()` executes the SQL statement:

```
try
{
    var command = connection.CreateCommand();
    command.CommandText = "INSERT INTO Orders (CustomerId, " +
        "OrderDate, ShipAddress, ShipCity, ShipCountry) " +
        "VALUES(@CustomerId, @OrderDate, @ShipAddress, @ShipCity, " +
        "@ShipCountry)";
    command.Parameters.AddWithValue("@CustomerId", order.CustomerId);
    command.Parameters.AddWithValue("@OrderDate", order.OrderDate);
```

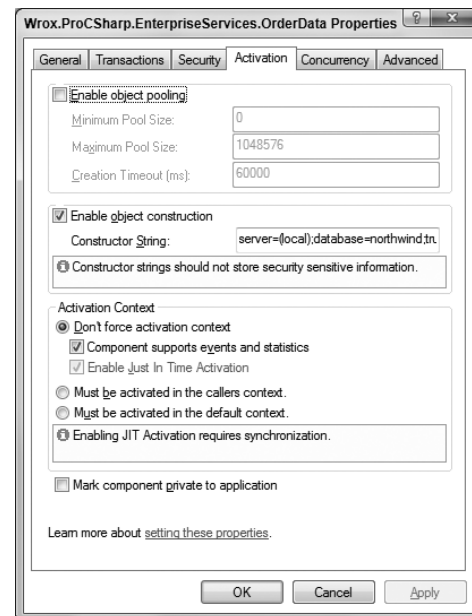


FIGURE 51-9


```

command.Parameters.AddWithValue("@ShipAddress", order.ShipAddress);
command.Parameters.AddWithValue("@ShipCity", order.ShipCity);
command.Parameters.AddWithValue("@ShipCountry", order.ShipCountry);

connection.Open();

command.ExecuteNonQuery();

```

Because `OrderId` is defined as an auto-increment value in the database, and this ID is needed for writing the Order Details to the database, `OrderId` is read by using `@@IDENTITY`. Then it is set to the `Order` object by calling the method `SetOrderId()`:

```

command.CommandText = "SELECT @@IDENTITY AS 'Identity'";
object identity = command.ExecuteScalar();
order.SetOrderId(Convert.ToInt32(identity));

```

After the order is written to the database, all order lines of the order are written using the `OrderLineData` component:

```

using (OrderLineData updateOrderLine = new OrderLineData())
{
    foreach (OrderLine orderLine in order.OrderLines)
    {
        updateOrderLine.Insert(order.OrderId, orderLine);
    }
}

```

Finally, regardless of whether the code in the `try` block was successful or an exception occurred, the connection is closed:

```

        finally
        {
            connection.Close();
        }
    }
}

```

The OrderLineData Component

The `OrderLineData` component is implemented similarly to the `OrderData` component. You use the attribute `ConstructionEnabled` to define the database connection string:



Available for
download on
Wrox.com

```

using System;
using System.EnterpriseServices;
using System.Runtime.InteropServices;
using System.Data;
using System.Data.SqlClient;

namespace Wrox.ProCSharp.EnterpriseServices
{
    [ComVisible(true)]
    public interface IOrderLineUpdate
    {
        void Insert(int orderId, OrderLine orderDetail);
    }

    [Transaction(TransactionOption.Required)]
    [EventTrackingEnabled(true)]
    [ConstructionEnabled(
        true, Default="server=(local);database=northwind;trusted_connection=true")]
    [ComVisible(true)]
    public class OrderLineData: ServicedComponent, IOrderLineUpdate

```

```

{
    private string connectionString;

    protected override void Construct(string s)
    {
        connectionString = s;
    }
}

```

code snippet NorthwindComponents/OrderLineData.cs

With the `Insert()` method of the `OrderLineData` class in this example, the `AutoComplete` attribute isn't used to demonstrate a different way to define the transaction outcome. It shows how to set the consistent and done bits with the `ContextUtil` class instead. The method `SetComplete()` is called at the end of the method, depending on whether inserting the data in the database was successful. If there is an error where an exception is thrown, the method `SetAbort()` sets the consistent bit to `false` instead, so that the transaction is undone along with all components participating in the transaction:

```

public void Insert(int orderId, OrderLine orderDetail)
{
    var connection = new SqlConnection(connectionString);
    try
    {
        var command = connection.CreateCommand();
        command.CommandText = "INSERT INTO [Order Details] (OrderId, " +
            "ProductId, UnitPrice, Quantity)" +
            "VALUES(@OrderId, @ProductId, @UnitPrice, @Quantity)";
        command.Parameters.AddWithValue("@OrderId", orderId);
        command.Parameters.AddWithValue("@ProductId", orderDetail.ProductId);
        command.Parameters.AddWithValue("@UnitPrice", orderDetail.UnitPrice);
        command.Parameters.AddWithValue("@Quantity", orderDetail.Quantity);

        connection.Open();

        command.ExecuteNonQuery();
    }
    catch (Exception)
    {
        ContextUtil.SetAbort();
        throw;
    }
    finally
    {
        connection.Close();
    }
    ContextUtil.SetComplete();
}
}

```

Client Application

Having built the component, you can create a client application. For testing purposes, a console application serves the purpose. After referencing the assembly `NorthwindComponent` and the assembly `System.EnterpriseServices`, you can create a new order with the static method `Order.Create()`. The `order.AddOrderLine()` method adds an order line to the order. `OrderLine.Create()` accepts product IDs, the price, and quantity to create an order line. In a real application, it would be useful to add a `Product` class instead of using product IDs, but the purpose of this example is to demonstrate transactions in general.

Finally, the serviced component class `OrderControl` is created to invoke the method `NewOrder()`:



```
var order = Order.Create("PICCO", DateTime.Today, "Georg Pippis",
                        "Salzburg", "Austria");
order.AddOrderLine(OrderLine.Create(16, 17.45F, 2));
order.AddOrderLine(OrderLine.Create(67, 14, 1));

using (var orderControl = new OrderControl())
{
    orderControl.NewOrder(order);
}
```

code snippet ClientApplication/Program.cs

You can try to write products that don't exist to the `OrderLine` (using a product ID that is not listed in the table `Products`). In this case, the transaction is aborted, and no data is written to the database.

While a transaction is active, you can see the transaction in the Component Services Explorer by selecting Distributed Transaction Coordinator in the tree view (see Figure 51-10).



You might have to add a sleep time to the `Insert()` method of the `OrderData` class to see the live transaction; otherwise, the transaction might be completed too fast to display.

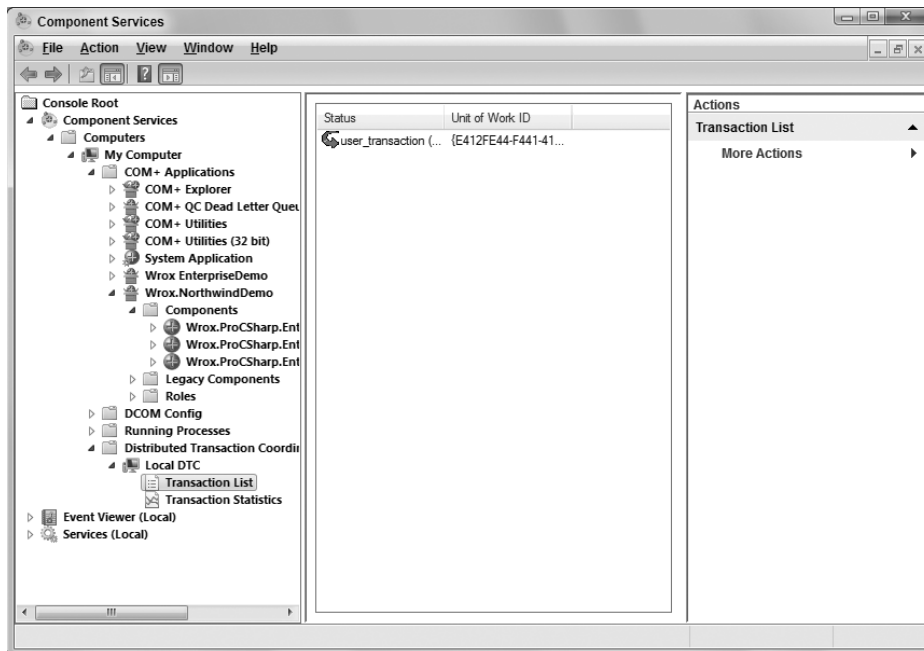


FIGURE 51-10



If you are debugging the serviced component while it is running inside a transaction, be aware that the default transaction timeout is 60 seconds for serviced components. You can change the default for the complete system in the Component Services Explorer by clicking My Computer, selecting Action Properties, and opening the Options tab. Instead of changing the value for the complete system, the transaction timeout can also be configured on a component-by-component level with the Transaction options of the component.

INTEGRATING WCF AND ENTERPRISE SERVICES

Windows Communication Foundation (WCF) is the preferred communication technology for .NET Framework 4. WCF is covered in detail in Chapter 43. .NET Enterprise Services offers a great integration model with WCF.

WCF Service Façade

Adding a WCF façade to an Enterprise Services application allows WCF clients to access the serviced components. Instead of using the DCOM protocol, with WCF you can have different protocols such as HTTP with SOAP or TCP with a binary formatting.

You can create a WCF façade from Visual Studio 2010 by selecting **Tools** ⇨ **WCF Service Configuration Editor**. Then select **File** ⇨ **Integrate** ⇨ **COM+ Applications**. Finally, select the COM+ Application Wrox.NorthwindDemo, the component Wrox.ProCSharp.EnterpriseServices.OrderControl, and the interface IOrderControl, as shown in Figure 51-11.



Instead of using Visual Studio to create the WCF façade, you can use the command-line utility comsvcsconfig.exe. You can find this utility in the directory <Windows>\Microsoft.NET\Framework\v4.0.

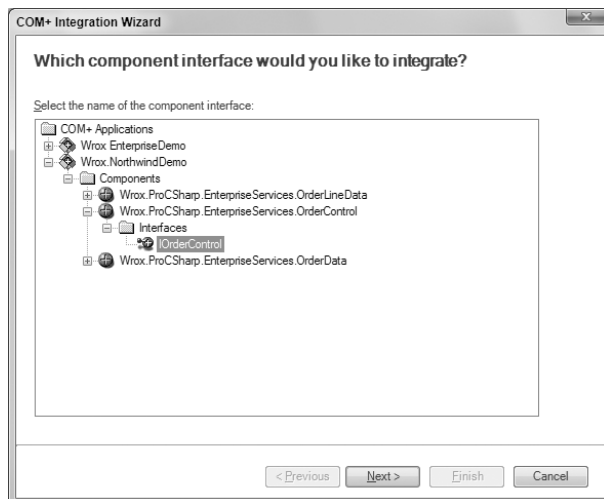


FIGURE 51-11

With the next dialog you can select all the methods from the interface `IOrderControl` that should be available to WCF clients. With the interface `IOrderControl` just one method, `NewOrder()`, is shown.

The next dialog, shown in Figure 51-12, allows configuration of the hosting options. With the hosting option, you can specify in which process the WCF service should run. When you select the COM+ hosted option, the WCF façade runs within the `dllhost.exe` process of COM+. This option can only be configured if the application is configured as a server application.

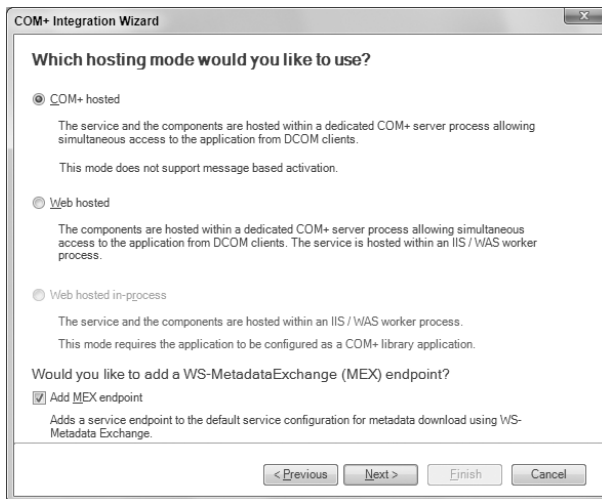


FIGURE 51-12

`ApplicationActivation(ActivationOption.Server)`. The Web hosted option specifies that the WCF channel listens inside a process of the IIS or WAS (Windows Activation Services) worker process. WAS is new since Windows Vista and Windows Server 2008. Selecting “Web hosted in-process” means that the library of the Enterprise Services component runs within the IIS or WAS worker process. This configuration is possible only if the application is configured as a library application: `ApplicationActivation(ActivationOption.Library)`.

Selecting the “Add MEX endpoint” option adds a Metadata Exchange (MEX) endpoint to the WCF configuration file, so that the client programmer can access the metadata of the service using WS-Metadata Exchange.



MEX is explained in Chapter 43.

With the next dialog shown in Figure 51-13 you can specify the communication mode to access the WCF façade. Depending on your requirements, if the client is accessing the service across a firewall or platform-independent communication is required, HTTP is the best choice. TCP offers faster communication across machines for .NET clients, and Named Pipes is the fastest option if the client application is running on the same system as the service.

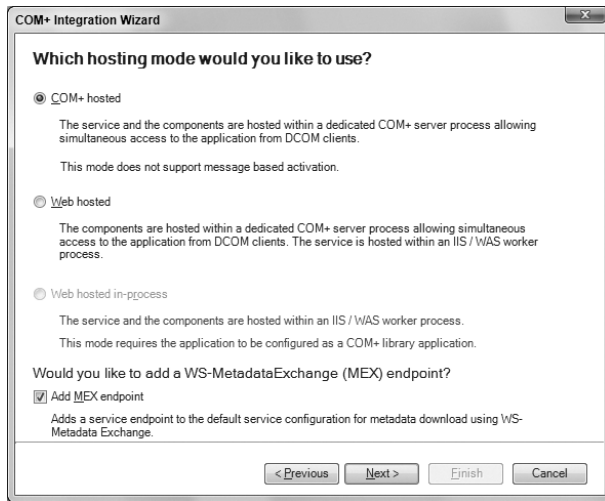


FIGURE 51-13

The next dialog requests information about the base address of the service that depends on the communication protocol selection, as shown in Figure 51-14.

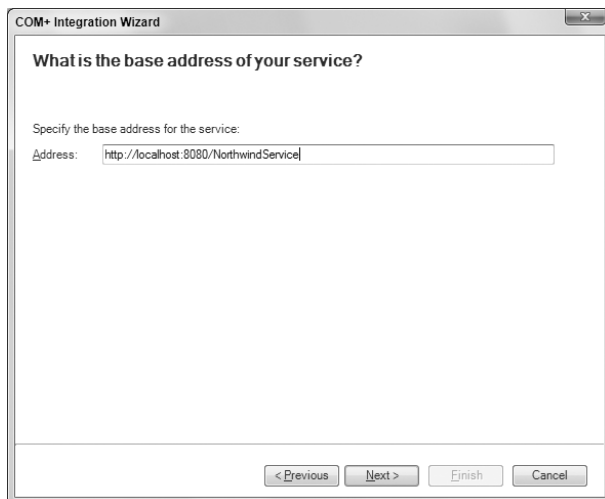


FIGURE 51-14

The last dialog shows the location of the endpoint configuration. The base directory for the configurations is <Program Files>\ComPlus Applications followed by the unique ID of the application. In this directory you can find the file `application.config`. This configuration file lists the behaviors and endpoints for WCF.

The <service> element specifies the exposed WCF service with the endpoint configuration. The binding is set to `wsHttpBinding` with a `comTransactionalBinding` configuration, so transactions can flow from the caller to the serviced component. With other network and client requirements, you can specify a different binding, but this is all covered in Chapter 43:

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="ComServiceMexBehavior">
          <serviceMetadata httpGetEnabled="true" />
          <serviceDebug />
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <bindings>
      <netNamedPipeBinding>
        <binding name="comNonTransactionalBinding" />
        <binding name="comTransactionalBinding" transactionFlow="true" />
      </netNamedPipeBinding>
      <wsHttpBinding>
        <binding name="comNonTransactionalBinding" />
        <binding name="comTransactionalBinding" transactionFlow="true" />
      </wsHttpBinding>
    </bindings>
    <comContracts>
      <comContract contract="{E1B02E09-EE48-3B6B-946F-E6A8BAEC6340}"
        name="IOrderControl"
        namespace=
          "http://tempuri.org/E1B02E09-EE48-3B6B-946F-E6A8BAEC6340"
        requiresSession="true">
        <exposedMethods>
          <add exposedMethod="NewOrder" />
        </exposedMethods>
      </comContract>
    </comContracts>
    <services>
      <service behaviorConfiguration="ComServiceMexBehavior"
        name="{196F39D0-4F47-454A-BC16-955C2C54B6F5},
          {A16C0740-C2A0-38C9-9FD3-7C583B3B42FA}">
        <endpoint address="IOrderControl" binding="wsHttpBinding"
          bindingConfiguration="comTransactionalBinding"
          contract="{E1B02E09-EE48-3B6B-946F-E6A8BAEC6340}" />
        <endpoint address="mex" binding="mexHttpBinding"
          contract="IMetadataExchange" />
        <host>
          <baseAddresses>
            <add baseAddress=
              "net.pipe://localhost/Wrox.NorthwindDemo/Wrox.ProCSharp.
                EnterpriseServices.OrderControl" />
            <add baseAddress=
              "http://localhost:8088/NorthwindService" />
          </baseAddresses>
        </host>
      </service>
    </services>
  </system.serviceModel>
</configuration>

```

Before you can start the server application you need to change the security to allow the user that runs the application to register ports for listening. Otherwise a normal user is not allowed to register a listener port. On Windows 7 you can do this with the `netsh` command as shown. The option `http` changes the ACL for the HTTP protocol. The port number and the name of the service are defined with the URL, and the user option specifies the name of the user that starts the listener service:

```
netsh http add urlacl url=http://+:8088/NorthwindService user=username
```

Client Application

Create a new console application named `WCFClientApp`. Because the service offers a MEX endpoint, you can add a service reference from Visual Studio by selecting **Project** ➤ **Add Service Reference . . .** (see Figure 51-15).



If the service is COM+ hosted, you have to start the application before you can access the MEX data. If the service is hosted inside WAS, the application is started automatically.

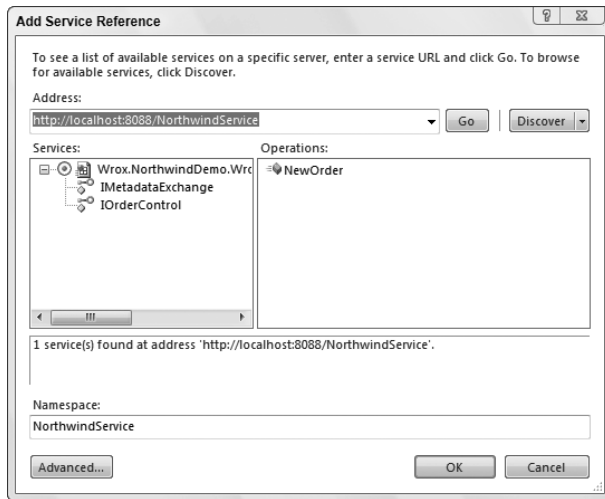


FIGURE 51-15

With the service reference a proxy class is created, the assemblies `System.ServiceModel` and `System.Runtime.Serialization` are referenced, and an application configuration file referencing the service is added to the client application.

Now you can use the generated entity classes and the proxy class `OrderControlClient` to send an order request to the serviced component:

```
static void Main()
{
    var order = new Order
    {
        CustomerId = "PICCO",
        OrderDate = DateTime.Today,
        ShipAddress = "Georg Pippis",
        ShipCity = "Salzburg",
        ShipCountry = "Austria"
    };
    var line1 = new OrderLine
    {
        ProductId = 16,
        UnitPrice = 17.45F,
        Quantity = 2
    };
    var line2 = new OrderLine
```



```
{
    ProductId = 67,
    UnitPrice = 14,
    Quantity = 1
}
OrderLine[] orderLines = { line1, line2 };
order.orderLines = orderLines;

var occ = new OrderControlClient();
occ.NewOrder(order);
}
```

SUMMARY

This chapter discussed the rich features offered by Enterprise Services, such as automatic transactions, object pooling, queued components, and loosely coupled events.

To create serviced components, you have to reference the assembly `System.EnterpriseServices`. The base class of all serviced components is `ServicedComponent`. In this class, the context makes it possible to intercept method calls. You can use attributes to specify the interception that will be used. You also learned how to configure an application and its components using attributes, as well as how to manage transactions and specify the transactional requirements of components by using the `[Transaction]` attribute. You've also seen how well Enterprise Services integrates with the new communication technology WCF, by creating a WCF façade.

This chapter showed how to use Enterprise Services, a feature offered by the operating system. The next chapter gives information on how to use another feature from the operating system that is used for communication: message queuing.

