Chapter 31

Serviced Components

Serviced components are Microsoft .NET Framework objects that run under Component Services and can leverage the full range of COM+ services, including just-in-time activation (JITA), automatic transactions, synchronization, object pooling, role-based security (RBS), and programmatic security. You can run a serviced component as a library component (in the client's process) or a server component (in a different process, possibly running on a different computer), even though a few COM+ services are available only in server libraries.

If you aren't familiar with serviced components and their benefits, you can read the whole story at *http://msdn.microsoft.com/library/en-us/cpguide/html/cpconwritingservicedcomponents.asp*. If you have already worked with serviced components, we're sure you'll find some interesting tips in this chapter.

Note One or more code examples in this chapter assume that the following namespaces have been imported by means of *Imports* (Visual Basic) or *using* (C#) statements:

```
System.Data.SqlClient
System.EnterpriseServices
```

31.1 COM+ transactions vs. ADO.NET transactions

Carefully consider the pros and cons of using transactional serviced components to implement COM+ transactions as opposed to using standard ADO.NET transactions.

Why: Implementing transactions in serviced components (COM+ transactions) offers several advantages, including the support for distributed databases, a higher degree of independence from the database, and a cleaner object-oriented design (for example, you can use attributes to select the transaction isolation level).

Why not: COM+ transactions use the Microsoft Distributed Transaction Coordinator (MS DTC). DTC-based transactions can be from 10 to 50 percent slower than ADO.NET transactions. When working with a single database server, you might decide to use ADO.NET transactions from inside a standard .NET class rather than encapsulating the database code in a serviced component.

More details: Another factor to consider when deciding which type of transaction to adopt is that serviced components running under Microsoft Windows 2000 can use only the Serializable isolation level; therefore, an ADO.NET transaction can give you more flexibility. Under Microsoft Windows Server 2003, you can use the Transaction attribute to select the actual isolation level (see rule 31.8).

In some cases, you can reduce the overhead of COM+ transactions from inside serviced components by turning off automatic enlistment of an ADO.NET connection. If you're using the Microsoft SQL Server .NET Data provider, you can disable automatic enlistment by setting the Enlist attribute to false in the connection string, as follows:

```
Data Source=.;Integrated Security=SSPI;Initial Catalog=Pubs;Enlist=false
```

31.2 Static members

Avoid public static members (*Shared* in Visual Basic) in types that inherit from System.EnterpriseServices.ServicedComponent.

Why: .NET serviced components support remote method invocation of their instance methods only. Also, COM+ interception doesn't work with static members; thus, the transactional context doesn't flow correctly through static methods calls.

31.3 Library vs. server components

Follow these rules when deciding whether to implement a library or a server COM+ component:

a. Server components can run remotely, on a computer other than the client's machine, and therefore can improve the application's scalability.

b. Server components can easily impersonate an identity other than the client's identity.

c. Server components live inside separate applications that can be restarted automatically under certain conditions.

d. If you don't need the security, scalability, and fault tolerance features of server COM+ components, use client components to achieve better performance.

e. All the arguments passed to and returned from a method defined in a server component must be either marked as serializable or derive from MarshalByRefObject.

f. Server components can run inside a Windows service.

g. Both library and server components must reside in strong-named assemblies. In addtion, server components must be registered in the GAC.

How to: You decide between a server or library COM+ component by marking the assembly with a suitable ApplicationActivation attribute (see example in rule 31.4).

31.4 Assembly-level attributes

Assign a value to the ApplicationName, ApplicationID, ApplicationActivation, and Description attributes for assemblies that contain serviced components.

Why: The ApplicationName attribute is the name that identifies the application in the Component Services administration snap-in; the Description attribute is used to describe the application itself. The ApplicationID attribute assigns an explicit ID to the application. (If omitted, this ID is

generated automatically when the component is registered.) The ApplicationName and ApplicationID attributes affect what you see in the General tab of the application's Properties window.

More details: An explicit ApplicationID value is especially useful for having multiple assemblies share the same COM+ application (and therefore the same server-side process), which in turn optimizes cross-component communication and marshaling. However, keep in mind that this attribute prevents you from using COM+ 1.5 partitions; thus, you must omit it if you plan to use partitions.

```
' [Visual Basic]
<Assembly: ApplicationName("BankMoneyMover")>
<Assembly: Description("Components for moving money between accounts")>
<Assembly: ApplicationID("F088FCFF-6FF0-496B-9121-DC9EB9DAEFFA")>
' This is a library COM+ component.
<Assembly: ApplicationActivation(ActivationOption.Library)>
' Assemblies containing serviced components must have a strong name.
<Assembly: AssemblyKeyFile("c:\codearchitects.snk")>

// [C#]
[assembly: ApplicationName("BankMoneyMover")]
[assembly: Description("Components for moving money between accounts")]
[assembly: ApplicationID("F088FCFF-6FF0-496B-9121-DC9EB9DAEFFA")]
// This is a library COM+ component.
[assembly: ApplicationActivation(ActivationOption.Library)]
// Assemblies containing serviced components must have a strong name.
[assembly: AssemblyKeyFile(@"c:\codearchitects.snk")]
```

Configuration attributes are important especially in the developing and test phase because they help to register the serviced component correctly on the first launch and therefore support XCOPY deployment (this is known as *dynamic* or *lazy registration*). However, most attributes related to serviced components are used only if the COM+ application doesn't exist yet. On the customer's site, the application might be first launched by a user without administrative privileges and the installation would fail. For this reason, you should always rely on the regsvcs tool to register the component.

The only attributes that are always read from the metadata in the component and that supersede the attributes in the COM+ catalog are JustInTimeActivation, AutoComplete, and ObjectPooling, plus the SecurityRole attribute when used at the method level. The ObjectPooling attribute in source code can enable or disable object pooling, but COM+ always uses the pool size defined in the COM+ catalog.

31.5 The ClassInterface attribute

Apply the ClassInterface attribute to all serviced components to make them expose a dual interface if you don't need to apply role-based security (RBS) at the method level.

Why: If you omit this attribute, the Component Services MMC snap-in doesn't list the component's individual methods. Also, late-bound calls from unmanaged clients would ignore the AutoComplete attribute, and you'd be forced to commit or abort the transaction explicitly by using code.

Why not: You can't apply this attribute when you need to apply method-level RBS, as explained in rule 33.17.

```
' [Visual Basic]
<ClassInterface(ClassInterfaceType.AutoDual)> _
Public Class MoneyMover
   Inherits ServicedComponent
   ...
End Class

// [C#]
[ClassInterface(ClassInterfaceType.AutoDual)]
public class MoneyMover : ServicedComponent
{
   ...
}
```

31.6 The JustInTimeActivation attribute

Mark serviced components with the JustInTimeActivation attribute.

Why: A JIT-activated component is more scalable and resource-savvy because the COM+ infrastructure instantiates it only when one of its methods is invoked and destroys it when the method completes, assuming that the method signals that the task has been completed or is marked with an AutoComplete attribute (see rule 31.7). Also, pooled objects should always be marked with the JustInTimeActivation attribute (see rule 31.11).

Why not: The JustInTimeActivation attribute should be used only for components that are designed to be used in a stateless fashion. You can't just add this attribute to a nontransactional component that you have already tested because the attribute changes the component's lifetime and, consequently, the way clients should use the component.

Another potential problem of JIT-activated components: the server must keep alive a wrapper of the component. Such a wrapper might take a significant amount of memory on the server; therefore, unnecessarily using this attribute can affect the application's performance and scalability negatively.

The JustInTimeActivation attribute is mainly useful to prevent clients from using a stateless component in an incorrect way. If you're using a nontransactional component, you can usually get better performance by omitting this attribute and letting the client release the object explicitly. If the client is itself stateless (for example, it's an ASP.NET Web Forms application), using this attribute is superfluous and might be avoided with nontransactional and nonpoolable components.

More details: A transactional type, that is, a class flagged with the Transaction attribute, is also implicitly a JIT-activated component. However, you should apply an explicit JustInTimeActivation attribute even to transactional types to make the code more readable and avoid problems if you later decide to remove the Transaction attribute.

```
' [Visual Basic]
<Transaction(TransactionOption.Required), JustInTimeActivation()> _
Public Class MoneyMover
   Inherits ServicedComponent
   ...
End Class

// [C#]
[Transaction(TransactionOption.Required)]
[JustInTimeActivation]
public class MoneyMover : ServicedComponent
{
   ...
}
```

31.7 The AutoComplete attribute

Control the outcome of a transaction by applying the AutoComplete attribute rather than by invoking the SetComplete or SetAbort methods. Explicitly throw exceptions if the transaction should be aborted and avoid catching exceptions (unless you rethrow them) when calling other components or methods exposed by .NET Framework objects.

Why: A serviced component should throw an exception whenever something goes wrong. Using the AutoComplete attribute helps you enforce this rule and makes your code more concise and easier to debug.

```
' [Visual Basic]
<AutoComplete()> _
Public Sub TransferMoney(ByVal accountID As Integer, ByVal amount As Decimal)
   ...
End Sub

// [C#]
[AutoComplete]
public void TransferMoney(int accountID, decimal amount)
{
   ...
}
```

31.8 The Transaction attribute

As a rule, use the default Serializable value for the Transaction attribute.

Why: Sticking to the default isolation level makes the component more easily reusable and avoids several potential problems.

Why not: You can usually achieve better performance by using a different isolation level. (You can set a nondefault value only in Microsoft Windows XP and Windows Server 2003 platforms.)

More details: The isolation level of a transaction is determined by the root component—that is, the first transactional component in the call chain. If this root component calls a child

component whose isolation level is equal to or higher than the root's isolation level, everything works smoothly; otherwise, the cross-component call will fail with an E_ISOLATIONLEVELMISMATCH error.

You can set the transaction support and the isolation level also from the Component Services MMC snap-in, as shown in Figure 31-1.

See also: See rules 31.1, 31.9, and 31.10 for exceptions to this rule.

```
' [Visual Basic]
<Transaction(TransactionOption.Required, _
 Isolation:=TransactionIsolationLevel.Serializable)> _
Public Class MoneyMover
   Inherits ServicedComponent
   ...
End Class

// [C#]
[Transaction(TransactionOption.Required, Isolation=TransactionIsolationLevel.Serializable)]
public class MoneyMover : ServicedComponent
{
   ...
}
```

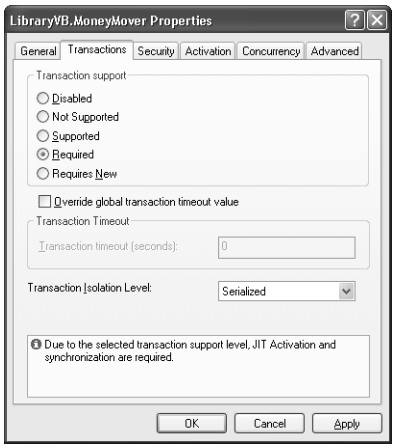

Figure 31-1 The Transactions page of a serviced component's Properties dialog box

31.9 Isolation level for nonroot components

Consider using the TransactionIsolationLevel.Any value as the isolation level for nonroot components.

Why: This special value forces the component to use the isolation level set by the component that is the root of the current transaction and offers a simple mechanism for making the component reusable in different situations.

```
' [Visual Basic]
<Transaction(TransactionOption.Supported, Isolation:=TransactionIsolationLevel.Any)> _
Public Class MoneyMover
   Inherits ServicedComponent
   ...
End Class

// [C#]
[Transaction(TransactionOption.Supported, Isolation=TransactionIsolationLevel.Any)]
public class MoneyMover : ServicedComponent
{
   ...
}
```

31.10 Types with methods that require different isolation levels

If a type exposes methods that require different isolation levels, consider creating a facade component that uses two (or more) types marked with different Transaction attributes.

Example: Let's say that you have one method that updates a database and requires the Serializable level, whereas another method performs a read operation for which a ReadCommitted level would be enough. If a component exposes both these methods, the best you can do is mark the component with a Transaction attribute that specifies a Serializable isolation level and accept the unnecessary overhead that results when the latter method is invoked. To avoid this overhead, you can create two additional classes: one that performs all the write operations at the Serializable level, and one that performs all read operations at the ReadCommitted level. The original component would have no Transaction attribute and would be responsible only for dispatching calls to one of the two types, depending on whether it's a write or a read operation.

31.11 Poolable objects

Override the CanBePooled method to ensure that the object is returned to the pool as soon as it has completed its job. Remember that pooled objects should be marked with the JustInTimeActivation attribute (see rule 31.5).

Why: Object pooling enables you to use resources effectively if clients create many objects and these objects take a significant time to initialize. In addition, pooling gives you the ability to configure the maximum number of objects that can be running at any given time. You need to employ the technique described in this guideline because serviced components aren't poolable by default.

Why not: You might decide not to use object pooling for objects that initialize very quickly. Also, object pooling shouldn't be used to implement a singleton model by forcing the maximum size of the pool to one.

How to: You can make a component poolable by marking it with an ObjectPooling attribute. You can also specify the minimum and/or the maximum number of objects in the pool, even though the values entered in the Component Services MMC snap-in have higher priority than those specified in the ObjectPooling attribute (see Figure 31-2). However, keep in mind that

the higher value you assign to the MinPoolSize property, the longer the first instantiation of the component will take.

```
' [Visual Basic]
<ObjectPooling(True, MinPoolSize:=4, MaxPoolSize:=20), _
 JustInTimeActivation()> _
Public Class MoneyMover
   Inherits ServicedComponent

   Protected Overrides Function CanBePooled() As Boolean
      Return True
   End Function
End Class

// [C#]
[ObjectPooling(true, MinPoolSize=4, MaxPoolSize=20)]
[JustInTimeActivation()]
public class MoneyMover : ServicedComponent
{
   protected override bool CanBePooled()
   {
      return true;
   }
}
```

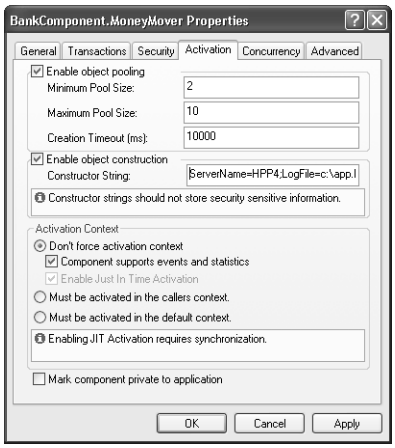

Figure 31-2 The Activation tab of the Properties page of a COM+ component enables you to change object pooling settings.

31.12 The ApplicationAccessControl attribute

Add an assembly-level ApplicationAccessControl attribute to enable COM+ role-based security and to enforce checks at the process and component level.

More details: In .NET Framework version 1.1, the COM+ security is enabled by default if the ApplicationAccessControl attribute is omitted; in version 1.0, COM+ security was disabled by default. Explicitly adding this attribute is recommended to improve readability.

```
' [Visual Basic]
<Assembly: ApplicationAccessControl(True, _
   AccessChecksLevel:=AccessChecksLevelOption.ApplicationComponent)>

// [C#]
[assembly: ApplicationAccessControl(true,
   AccessChecksLevel=AccessChecksLevelOption.ApplicationComponent)]
```

31.13 The authentication level

In server applications, set the authentication level to Privacy, unless it is safe to use less severe settings.

More details: The authentication level of a library component is inherited from the client process. The Authentication property of the ApplicationAccessControl attribute lets you decide how a server component authenticates data coming from the caller. Setting the authentication level to Privacy ensures that COM+ authenticates the caller's credentials and encrypts each data packet, thus providing the most secure type of authentication but affecting performance negatively. If data sniffing isn't an issue, you might decide to use a more efficient setting, such as Connect (authenticates credentials only when the connection is established), Call (authenticates credentials on each call), Packet (authenticates credentials and ensures that all packets are received), or Integrity (authenticates credentials and ensures that no data packet has been modified).

```
' [Visual Basic]
<Assembly: ApplicationAccessControl(True, _
   AccessChecksLevel:=AccessChecksLevelOption.ApplicationComponent, _
   Authentication:=AuthenticationOption.Privacy)>

// [C#]
[assembly: ApplicationAccessControl(true,
   AccessChecksLevel=AccessChecksLevelOption.ApplicationComponent,
   Authentication=AuthenticationOption.Privacy)]
```

Remember that the actual authentication level used by a COM+ component depends also on the authentication level set by the client. When the component's and the client's authorization levels differ, COM+ uses the higher level of the two. If the client is an ASP.NET application, you can configure its authentication level by means of the comAuthenticationLevel attribute in the <processModel> element in machine.config.

31.14 The impersonation level

In server applications, set the impersonation level to Identify, unless you need to enable impersonation or delegation.

More details: The impersonation level of a library COM+ component is inherited from the client process and can't be changed. The ImpersonationLevel property of the ApplicationAccessControl attribute lets you decide whether another component called by the current COM+ component can discover the identity of the caller and can impersonate the caller when calling

services running on different computers. The available settings are Anonymous (the component is unaware of the caller's identity and can't access local or remote resources on the caller's behalf), Identify (the component can determine the caller's identity), Impersonate (the component impersonates the caller when accessing local resources, or even resources on a different computer if the caller resides on the same machine as the component), and Delegate (the component impersonates the caller when accessing resources on the local computer as well as any remote server; this setting requires the Kerberos authentication services and that the Active Directory directory service is configured on both the client and the server machine).

When using the Identity setting, the called COM+ component can use the SecurityCallContext object to determine the caller's identity (the DirectCaller property) and whether the caller is in a given role (the IsCallerInRole method), but the component can't impersonate the caller when accessing a database or other resources, either on the same machine or on a remote server. Because the trusted subsystem is recommended in multitiered architectures (see rule 29.27), impersonation and delegation are usually neither necessary nor desirable.

```
' [Visual Basic]
<Assembly: ApplicationAccessControl(True, _
   AccessChecksLevel:=AccessChecksLevelOption.ApplicationComponent, _
   Authentication:=AuthenticationOption.Privacy, _
   ImpersonationLevel:=ImpersonationLevelOption.Identify)>

// [C#]
[assembly: ApplicationAccessControl(true,
   AccessChecksLevel=AccessChecksLevelOption.ApplicationComponent,
   Authentication=AuthenticationOption.Privacy,
   ImpersonationLevel=ImpersonationLevelOption.Identify)]
```

31.15 The ComponentAccessControl attribute

Add a class-level ComponentAccessControl attribute to enable security checks for that component.

More details: This attribute is ignored if access checks are enabled only at the application level (see rule 31.12).

```
' [Visual Basic]
<ComponentAccessControl(True)> _
Public Class MoneyMover
Inherits ServicedComponent
   ...
End Class

// [C#]
[ComponentAccessControl(true)]
public class MoneyMover : ServicedComponent
{
   ...
}
```

31.16 The SecurityRole attribute

Add one or more assembly-level SecurityRole attributes that define all the user roles recognized by the application. Always include a SecurityRole attribute that adds the Everyone user to the Marshaler role if you plan to enable method-level security.

More details: You can apply the SecurityRole attribute at the assembly, class, and method level. When applied at the assembly level, it defines which users can activate any component in the application, provided that you've applied an ApplicationAccessControl(true) attribute. When applied at the class level, it defines which users can call any method in that class, provided that you have applied a ComponentAccessControl(true) attribute to the class. You apply the SecurityRole attribute to individual methods only when you enable security at the method level, as explained in rule 31.17.

The registration process adds all the roles that you specify in SecurityRole attributes in the COM+ catalog. By default, these roles contains no users, but you can add the Everyone user to a role by passing true in the second argument (which corresponds to the SetEveryoneAccess property). Users other than Everyone can be added to a role only by means of the COM+ explorer or by using an administrative script.

```
' [Visual Basic]
' Accountants can launch this application; all users are in this role.
<Assembly: SecurityRole("Accountants", True)>
' Create the Managers role, but don't add any users to it.
<Assembly: SecurityRole("Managers")>
' Prepare the application for security at the method level.
<Assembly: SecurityRole("Marshaler", True)>

<SecurityRole("Readers", True, Description:="Users who can read")> _
Public Class MoneyMover
Inherits ServicedComponent
   ...
End Class

// [C#]
// Accountants can launch this application; all users are in this role.
[assembly: SecurityRole("Accountants", true)]
// Create the Managers role, but don't add any users to it.
[assembly: SecurityRole("Managers")]
// Prepare the application for security at the method level.
[assembly: SecurityRole("Marshaler", true)]

[SecurityRole("Readers", true, Description="Users who can read")]
public class MoneyMover : ServicedComponent
{
   ...
}
```

31.17 COM+ role-based security at the method level

Follow these steps to correctly enable role-based security (RBS) at the method level:

1. Ensure that you marked the assembly with an ApplicationAccessControl(true) attribute (see rule 31.12) and the serviced component class with a ComponentAccessControl(true) attribute whose AccessChecksLevel property is set to ApplicationComponent (see rule 31.15).
2. Add an assembly-level SecurityRole attribute that adds the Everyone user to the Marshaler role (see rule 31.16).
3. Mark the serviced component class with the SecureMethod attribute.
4. Define the methods to be secured in a separate interface and have the serviced component class implement the interface.
5. Apply the SecurityRole attribute to methods in the class, specifying which role can call which method, or use the MMC snap-in to perform this step from the user interface as shown in Figure 31-3. (Alternatively, you can add the SecurityRole attribute at the class level to configure all methods in the class for that role.)

```
' [Visual Basic]
<Assembly: ApplicationAccessControl(True, _
   AccessChecksLevel:=AccessChecksLevelOption.ApplicationComponent)>
<Assembly: SecurityRole("Marshaler", True)>

Public Interface IMoneyMover
   Sub MoveMoney(ByVal accountID As String, ByVal amount As Decimal)
End Interface

<ComponentAccessControl(True), SecureMethod()> _
Public Class MoneyMover
Inherits ServicedComponent
   Implements IMoneyMover

   <SecurityRole("Accountants", True)> _
   Public Sub MoveMoney(ByVal accountID As String, ByVal amount As Decimal) _
      Implements IMoneyMover.MoveMoney
      ...
   End Sub
End Class

// [C#]
[assembly: ApplicationAccessControl(true,
   AccessChecksLevel=AccessChecksLevelOption.ApplicationComponent)]
[assembly: SecurityRole("Marshaler", true)]

public interface IMoneyMover
{
   void MoveMoney(string accountID, decimal amount);
}
```

```
[ComponentAccessControl(true)]
[SecureMethod()]
public class MoneyMover : ServicedComponent, IMoneyMover
{
   [SecurityRole("Accountants", true)]
   public void MoveMoney(string accountID, decimal amount)
   {
      ...
   }
}
```

Notice that the SecureMethod attribute at the class level isn't strictly required if you use a SecurityRole attribute at the method level, but we recommend that you keep both attributes for increased readability.

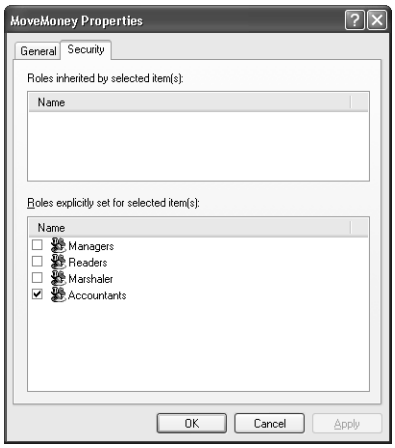

Figure 31-3 You can enforce which roles can call the component using the Security page in the Properties dialog box of individual methods.

31.18 Programmatic security

Use the SecurityCallContext object to perform programmatic security and always explicitly test that the IsSecurityEnabled property is true. Don't use the ContextUtil class to implement programmatic security.

Why: The SecurityCallContext object exposes a more complete set of security-related methods than the ContextUtil class. However, IsCallerInRole and other methods return true even if role-based security (RBS) is disabled; thus, you should always explicitly check that RBS is enabled by testing the IsSecurityEnabled property, as follows:

```
' [Visual Basic]
Dim scc As SecurityCallContext = SecurityCallContext.CurrentCall
If Not scc.IsSecurityEnabled Then
   Throw New Exception("This method requires role-based security")
ElseIf scc.IsCallerInRole("Managers") Then
   ...
End If
```

```
// [C#]
SecurityCallContext scc = SecurityCallContext.CurrentCall;
if ( ! scc.IsSecurityEnabled )
   throw new Exception("This method requires role-based security");
else if ( scc.IsCallerInRole("Managers") )
    ...
```

31.19 Component identity

Run a server COM+ component under the identity of a least-privileged specific account. Never run the component under the interactive user's identity, and avoid using predefined system accounts.

Why: Running the component under the identity of the interactive user is OK only during the development phase because this setting enables you to display message boxes and other diagnostic messages. In real applications, you should always define an ad-hoc account that has only the privileges that the application strictly requires, for example, access to certain directories and registry keys. This technique gives you more granular security than the one offered by predefined system accounts such as Local Service, Network Service, and Local System.

31.20 Disposing of a serviced component

The client application should dispose of all serviced components that aren't JIT-activated as soon as it is finished with them. The recommended way to do so is by calling the component's Dispose method, rather than by means of the ServicedComponent.DisposeObject static method.

```
' [Visual Basic]
' *** OK
ServicedComponent.DisposeObject(obj)
' *** Better
obj.Dispose()

// [C#]
// *** OK
ServicedComponent.DisposeObject(obj);
// *** Better
obj.Dispose();
```

For performance reasons, a serviced component should never implement the Finalize method because such a method would be called through reflection. Instead, override the protected Dispose(disposed) method and place all finalization code there.

31.21 WebMethod attributes in serviced components

Never mark a public method of a serviced component with a WebMethod attribute to make it callable through the Web service infrastructure.

Why: Both serviced components and Web services offer remote clients the ability to call methods in a .NET component. However, these two remoting technologies might conflict with each other because of the way they enable the transaction and context flow from the client and the component; therefore, you should never mix the two techniques in the same component.