

Expert ASP.NET 2.0

Advanced Application Design



Dominic Selly
Andrew Troelsen
Tom Barnaby

Expert ASP.NET 2.0 Advanced Application Design

Copyright © 2006 by Dominic Selly, Andrew Troelsen, and Tom Barnaby

Lead Editor: Ewan Buckingham

Technical Reviewers: Robert Lair, Rakesh Rajan, Victor Garcia Aprea, Fernando Simonazzi,
Hernan De Lahitte

Contributors: Dave Zimmerman, David Schueck, George Politis

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis,
Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser

Associate Publisher: Grace Wong

Project Manager: Beckie Stones

Copy Edit Manager: Nicole LeClerc

Copy Editor: Lauren Kennedy

Assistant Production Director: Kari Brooks-Copony

Production Editor: Kelly Winkquist

Compositor: Dina Quan

Proofreader: Nancy Sixsmith

Indexer: Broccoli Information Management

Artist: Wordstop

Interior Designer: Van Winkle Design Group

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit <http://www.springeronline.com>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Source Code section.



Enterprise Services

The hosting environment that was originally shipped as Microsoft Transaction Server (MTS) has a lot of names nowadays. Some call it Component Services. Some call it Enterprise Services. Some prefer the brevity of just COM+.

Whatever you call it, Component Services is the original aspect-oriented application server for Windows. Support for Component Services has been extended into the .NET Framework, so even though it's still a COM-based technology, you can create types in .NET that can be configured and hosted in the COM+ environment.

In this chapter we'll take a look at the features provided by this hosting environment, and then examine what you do exactly to create .NET types that can benefit from this feature set.

Component Services

Component Services provides a hosting environment and configuration registry for exposing software assets as services. Originally called Microsoft Transaction Server, these services were designed to be exposed in an RPC manner via DCOM. Over time, it became as common (if not more common) to leverage configured components from Internet Information Server (IIS). This was the standard architecture for ASP-based applications. Creating COM types and hosting them as configured services in COM+ provided many benefits, including compiled code (as opposed to interpreted script) and a type system (as opposed to one type: the VBScript Variant). Figure 7-1 shows the original common architectures.

Many powerful features of Component Services worth leveraging from managed code still exist. Support for running managed code within the COM environment of COM+ is accomplished via highly specialized interoperability code that has been added to the Component Services infrastructure. Because of this, not just any .NET type can be configured and hosted in this environment. Not only do you need to make special considerations for the design of these types, but the types must also inherit from a special base class and adhere to special run-time behaviors.

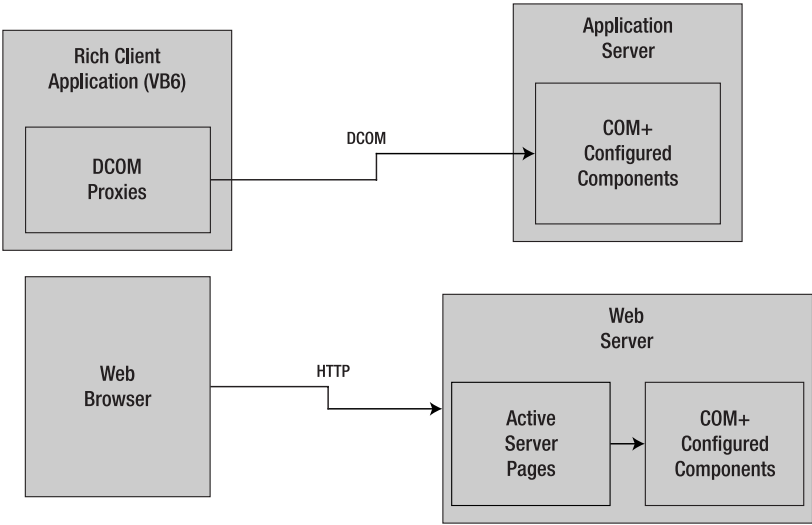


Figure 7-1. *The original common architectures for Component Services*

We'll take a look at how support for Component Services is grafted onto the .NET Framework over the course of the chapter. First, let's take a quick look at why you would be tempted to create configured components in the first place. Table 7-1 lists some of the features of COM+.

Table 7-1. *Features You Can Leverage from the Components Services Hosting Environment*

COM+ Feature	Meaning in Life
Thread Management	COM+ allocates thread pools to manage higher loads and service many requests concurrently. Combined with Just-in-Time Activation (JITA) and object pools, COM+ can greatly reduce the overhead of object instantiation and destruction, and can effectively manage object lifetimes in a highly scalable environment.
Transaction Management	Leveraging the features of the Microsoft Distributed Transaction Coordinator (MSDTC), types can be made to have their work participate in transactions declaratively. That is, transactional behavior becomes an aspect of the type, in many cases eliminating the need for special considerations while coding. Transactions can also be managed across disparate servers, even across different database vendors.
Queued Components	This feature set creates a perfect layer of abstraction between a developer and Message Queuing (MSMQ). Method calls become messages, benefiting your code by making your method calls asynchronous, which provides peak load balancing and guaranteed delivery. All of the details of creating a message, putting it in a queue, and processing it on the receiver are managed by the hosting environment.
Security	Applications, classes, and even methods can be declared as requiring the executing user be in a predefined role. This is declarative security, eliminating the need to modify imperative code to meet your authorization requirements.

These services are all provided with the concept of a call context. Objects that share run-time requirements will share context; objects with different run-time requirements will be created in different contexts. Contexts provide an interception boundary for the hosting environment. As code from a method in type A calls into code of a method in type B, execution does not immediately move from A to B. COM+ code intercepts the call, and additional work is done before execution moves to the code in type B. This interception boundary does things like enable COM+ to retrieve an object of type B from the pool to service the request. When the method call is done, the context interception code is fired again before control returns to the instance of A. COM+ can, at this point, put the object back into the pool, as would be the case with JITA, for example.

While contexts are invisible to the developer consuming these services, the presence of contexts in COM+ is ubiquitous; they are the mechanism via which all of its services are implemented. They also provide an additional layer of overhead, which you should always consider before you make the decision to move to COM+. We'll discuss some of these considerations in detail later in the chapter. First, let's take a look at what you have to do specifically to your managed code so that it will play nicely within the Component Services environment.

COM+ in .NET

Before we dig into the .NET-specific bits, let's take a minute to look at some terms listed in Table 7-2.

Table 7-2. *Names and Titles Used to Talk about COM+*

Term	Meaning in Life
Microsoft Transaction Server (MTS)	An application hosting environment that provides services to components via aspects, determining the behavior of the components with declarations rather than requiring imperative code. This is the predecessor to COM+.
COM+	The name given to the environment for hosting configured components. COM+ has some new features that were not present in MTS. It could have been called "MTS version 2.0", but Microsoft renamed it COM+ instead.
Component Services	Another name for COM+. This is the name given to the Microsoft Management Console (MMC) snap-in used for creating and configuring COM+ applications, and so has become another name for COM+.
Configured Component	When you register a class into COM+, it is said to be a configured component. The act of installing it into COM+ is the act of configuring it. This is true for COM types as well as .NET types, although you use different means to configure each of these types of components.
Enterprise Services	The set of features baked into the .NET Framework that enable you to create managed types that can live in COM+ (aka that can be configured). The namespace of the types supporting these features is <code>System.EnterpriseServices</code> , and so the term is frequently used to describe managed COM+ components.
Serviced Component	This is the name of a type built into the .NET Framework Class Library (the full name is <code>System.EnterpriseServices.ServicedComponent</code>). It is, therefore, used frequently to refer to a specific managed type designed to be hosted under Component Services (aka "Is the CustomerService object a Serviced Component?").

The managed functionality of COM+ is exposed in the .NET Framework via the assembly named `System.EnterpriseServices` (which lives in `System.EnterpriseServices.dll`). This assembly is part of the Framework class library, but you still have to explicitly add a reference to it from the project you'll be creating configured components in (see Figure 7-2).

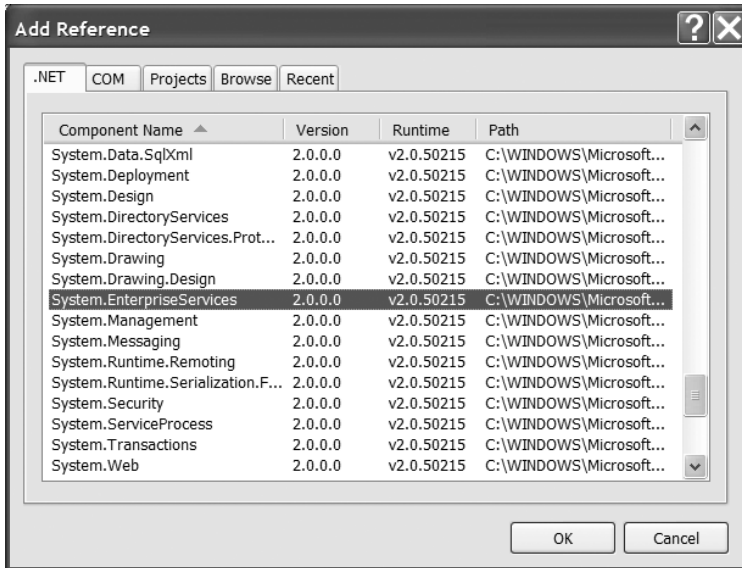


Figure 7-2. Adding a reference to `System.EnterpriseServices`

This will make the `ServicedComponent` class available to your class library project. All managed components that you want to configure to run under COM+ must use this type as their base class. Here's a simple implementation of a Serviced Component. (You can find this type in the `Serviced` project of the `Code07` solution.)

```
// All the important enterprise service types are contained here.
using System.EnterpriseServices;
```

```
namespace CarLibrary
{
    // Set the transaction mode to "supported"
    [Transaction(TransactionOption.Supported)]
    public class CarService : ServicedComponent
    {
        // If method raises exception, tx is automatically aborted.
        [AutoComplete(true)]
        public void InsertCar(DataSet carData)
        {
            // Insert the car data into the database
        }
    }
}
```

This type will now be hosted in Component Services, by virtue of the fact that it inherits from the `ServicedComponent` base class. It's configured to leverage transactions within Component Services. This is done via attributes applied to the type at the class and method level: `Transaction` and `AutoComplete`. These attributes determine the default configuration of the component when it's registered into COM+. We'll cover these attributes in detail as we examine specific functional areas of Component Services, for now just realize the *default* configured behavior of Serviced Components is always determined declaratively by .NET attributes (see Figure 7-3).

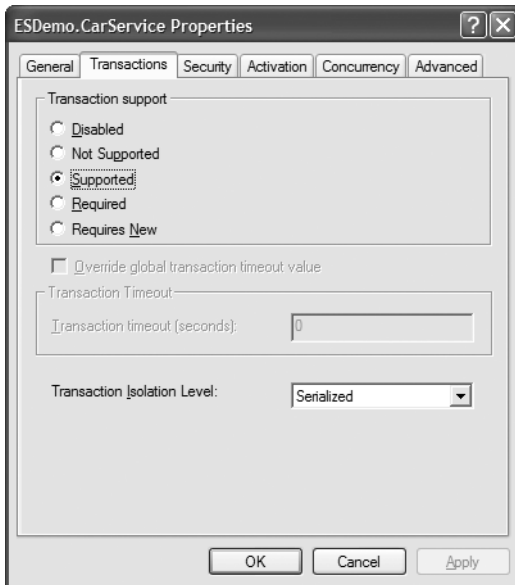


Figure 7-3. *The configuration of the CarService type as determined by its attributes*

The `ServicedComponent` base class deals with the underlying details of hosting a managed component in a COM environment. This is interoperability between .NET and COM, but it is not the same type of “COM Interop” you’re used to when, for example, you use the Office interop assemblies or your own COM proxy created with `tlbimp.exe`. This is more specialized interop code, designed and optimized specifically for interacting with Component Services. While its performance is better than standard COM Interop, it’s still a layer of abstraction, which will invariably cause a performance hit you must consider before you decide to adopt Enterprise Services. Keep in mind that when your Serviced Component is registered in COM+, a type library will be generated and the registry will be populated with information about it. Although this happens automatically behind the scenes, it’s important to realize you have these dependencies on traditional COM infrastructure.

Usually, the features that are needed from Component Services will offset the performance hit your application will incur by leveraging Enterprise Services. Perhaps you have scalability concerns as traffic increases to your application, and you know object pooling will help address it. Remember, many of the nonfunctional requirements you have to meet in your applications need to be done at the expense of performance. For scalability, moving your code

into Component Services by adopting Enterprise Services will make the application more scalable so it can deal with a higher number of concurrent users, but the experience of a single user will not be as fast.

Frequently it's the distributed transaction features that drive the adoption of Enterprise Services. If your application has complex transactional requirements, the management and services provided by COM+ will certainly justify the loss in performance incurred by using this COM environment. There's likely no way you could code this yourself and have it perform any better, much less be as reliable (or even work, for that matter).

You can determine most of the details of configuration under COM+ using .NET attributes to decorate your Serviced Components. There are some instances, though, when you need write code to interact with the hosting environment. For this interaction, there is the ContextUtil class, and its stable of static members. A partial list of them is shown in Table 7-3.

Table 7-3. *Static Members of the ContextUtil Type Used for Interacting with the COM+ Hosting Environment*

Static Member of ContextUtil	Meaning in Life
DeactivateOnReturn	Set to true when using JITA to return the instance to the pool at the end of a method call.
EnableCommit, DisableCommit	Enables your component to vote on the outcome of a transaction.
MyTransactionVote	Another way to vote on the outcome of a transaction.
GetNamedProperty SetNamedProperty	Access to the Shared Property Manager.
IsSecurityEnabled	Boolean indicated if security is in use for the current call context.
IsCallerInRole	Checks to see if a user is in a specified COM+ role.
IsInTransaction	Boolean indicating whether the work being done is transactional or not.

We'll take a look at more details of these ContextUtil members as they're relevant in the discussion of COM+ features that follows. Serviced Components must also always be strongly named.

COM+ Applications

Classes are deployed into COM+ using the abstraction of an *application*. A COM+ application can be thought of as simply an aggregation of configured classes that share run-time requirements. Once configured within COM+, these classes are also called *components*.

Components are aggregations of interfaces, and interfaces are aggregations of methods. Aspects of the run-time behavior that can be controlled at the application level will be shared across all components, all the way down to the method level. Some aspects can be added or overridden at each level in the hierarchy.

The most important configuration aspects at the application level are security and activation. *Security* controls what identity the components run under. When deciding what components should be grouped in an application, consider that cross-application calls can cross a security boundary, so you should logically group them for optimum performance.

Activation is the other prominent aspect. This aspect controls whether components are created in their own process, or whether they are created in the process of their caller. Applications created in their own process are called *server applications*, and while calling these components incurs the performance hit of crossing a boundary, you also gain the benefits of isolation. This can affect pool allocation and the identity of the process.

Applications created in the process of the caller are called *library applications*. They lose the capability to specify their run-time identity, as they will run under the identity of their caller. They will also have pools created for each application from which they're invoked (see Chapter 8 for more details of library versus server applications).

When you're creating Serviced Components, you can control these aspects of a COM+ application using assembly level attributes (these attributes can be found in `AssemblyInfo.cs` in the Serviced project).

```
[assembly: ApplicationName("Serviced")]
[assembly: ApplicationAccessControl(false)]
[assembly: ApplicationActivation(ActivationOption.Library)]
```

These attributes are then read via reflection and applied when the component is being configured. We'll examine configuration more closely after we look at the specific features you can leverage from within Component Services.

Just-In-Time Activation

This feature (abbreviated as JITA), enables an instance of an object to survive for the span of only a single method call. Even if a consumer of this type holds a reference to an instance of it for a long period of time, instances will only be created when the consumer actually calls a method.

JITA is configured on a class using the `JustInTimeActivation` attribute, as in the following class declaration.

```
[JustInTimeActivation(true)]
public class JITA : ServicedComponent
{
    //Class Implementation
}
```

The only other thing necessary to have COM+ destroy the object after a call is to apply the `DeactivateOnReturn` attribute. To illustrate the effect JITA has on object lifetimes, examine it via this simple service method. (This class can be found in the Serviced project of the Code07 solution.)

```
//[JustInTimeActivation(true)]
public class JITA : ServicedComponent
{
    private DateTime m_CreateStamp;
    public JITA()
    {
        m_CreateStamp = DateTime.Now();
    }
}
```

```

    public DateTime GetCreateStamp()
    {
        //ContextUtil.DeactivateOnReturn = true;
        return m_CreateStamp;
    }
}

```

Notice the JITA specific code is commented out. Now exercise this code with the following loop, and examine the output it generates. (You can find test code in the TestHarness project of the Code07 solution.)

```

static void Main(string[] args)
{
    JITA j = new JITA();

    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine(j.GetCreateStamp());
        Thread.Sleep(3000);
    }
    Console.ReadLine();
}

```

With the JITA attribute commented out, the dates on the output all match (see Figure 7-4). This makes sense, because the consumer is holding a reference to the same instance across all calls, and so the object is only created a single time.

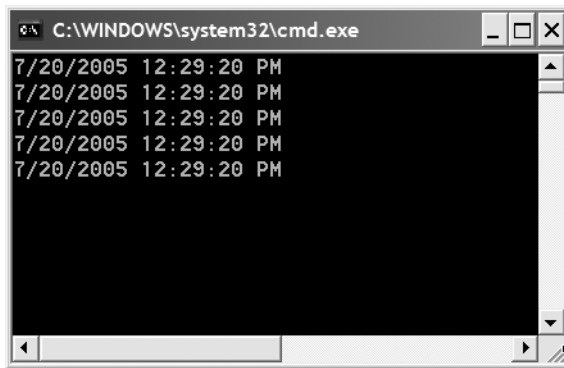


Figure 7-4. *The timestamp matches across all method calls without JITA.*

Look at what happens if you remove the comments around the JITA-specific code and rerun the client (see Figure 7-5).

Now, obviously the churn involved in object creation and destruction will, in most cases, consume the benefit gained by not keeping extraneous instances around between a client's method calls. For this reason, JITA usually makes the most sense when it's combined with object pooling. We'll take a look at object pooling in the next section.

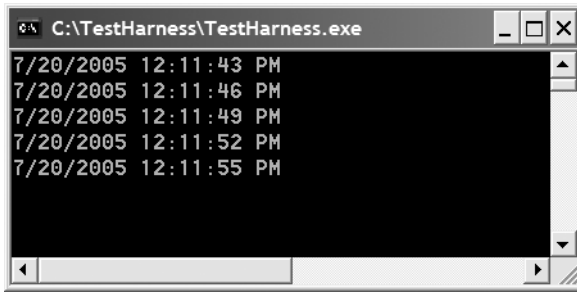


Figure 7-5. With JITA, each iteration of the loop is actually calling a new instance of the service object.

The goal of JITA is to optimize the efficiency of stateless components. Since classes designed to be used within COM+ should generally be stateless, JITA has broad applicability in this environment. Enabling this feature removes control of the object lifetime from the client, and puts the server component in control of its own lifetime. This is going to increase scalability if you have clients that are holding references to your components, even when they should be creating instances late and releasing them as soon as possible (a best practice in distributed, stateless programming environments).

It could be argued then, that if your client is stateless (like it is when the “client” is an ASP.NET web application), JITA is not needed because the lifetime will only ever last as long as the lifetime of the Web Form holding the reference. This is theoretically true, but even if your client is stateless, enabling JITA can guard against bad coding practices, like putting a reference to a COM+ component into the ASP.NET web cache. In larger environments where you may not necessarily be in a position to review the code that’s consuming your components, JITA can still be worthwhile.

The only time JITA should not be considered is when your component is maintaining state information across method calls. When this is necessary, the lifetime of the component must be managed by the client, and JITA-enabling this component will cause the state information to be lost.

We’ll be looking at COM+ transactions in a following section. It’s worth noting that enabling transactions on your type automatically causes it to be JITA-enabled.

Object Pooling

A pool of objects increases the scalability of an application by avoiding expensive object instantiation and destruction overhead, and it is able to service the requests of many times more clients than can be served with an instance per client. Coupling pools with JITA can dramatically increase the load your application can withstand.

A *pool* is nothing more than a number of active instances of a type that COM+ maintains in memory, and then dynamically allocates as clients request instances of the type. With JITA configured, these allocations occur on a *per method call* basis. When the method is finished executing, the instance is returned to the pool to service the next request. Not only are instances of the type more readily available, but also precious resources are saved by not instantiating an instance per client reference, and by avoiding the expensive process of allocating additional blocks of memory to hold the instance.

For this to work, the object *must be stateless*. Any field-level information designed to be maintained across method calls will not necessarily be maintained. These types must be entirely autonomous at the method level. They need to accept all of the parameters required to do their work, do the work within the method call, and release any resources used to do the work before returning results to the caller.

Pooling behavior is controlled with the `ObjectPooling` attribute, seen as follows. (You can find this class in the `Serviced` project of the `Code07` solution.)

```
[ObjectPooling(5, 500)]
public class Poolable : ServicedComponent
{
    public DataSet GetSomeData(string sql)
    {
        SqlConnection cn = new SqlConnection(ConnStr);
        SqlCommand cm = new SqlCommand(sql, cn);
        DataSet ds = new DataSet();

        new SqlDataAdapter(cm).Fill(ds);
        return ds;
    }
    protected override CanBePooled()
    {
        return true;
    }
}
```

The `ObjectPooling` attribute controls the default configuration of the component when it's registered with COM+ (see Figure 7-6). As objects are deactivated, the COM+ runtime calls the `CanBePooled` method to verify that it has permission to return the instance to the pool. This method returns `false` from the base class, so you need to override it and return `true` in order to get instances into the pool.

Pooling should always be used in combination with JITA, for highly available instances that are returned to the pool after each method completes.

```
[ObjectPooling(5, 500)]
[JustInTimeActivation(true)]
public class Poolable : ServicedComponent
{
    public DataSet GetSomeData(string sql)
    {
        SqlConnection cn = new SqlConnection(ConnStr);
        SqlCommand cm = new SqlCommand(sql, cn);
        DataSet ds = new DataSet();

        cn.Open();
        new SqlDataAdapter(cm).Fill(ds);
        cn.Close();
    }
}
```

```

        ContextUtil.DeactivateOnReturn = true;
        return ds;
    }
}

```

Notice that in this case, we've added not only the `JustInTimeActivation` attribute back on to the class definition, but also added a call to `DeactivateOnReturn` into the method body, to ensure COM+ knows the instance can be returned to the pool when the method call is complete.

Finally, you can use object pooling to throttle access to a limited resource. By minimizing the maximum pool size, you can control the number of concurrent requests that can be processed by the pooled object. For example, you might have a document management system with a ten-connection license. In order to avoid having more than ten concurrent connections, you can pool the object with a maximum pool size of ten, and a peak in load will be serialized after the tenth instance is served from the pool.

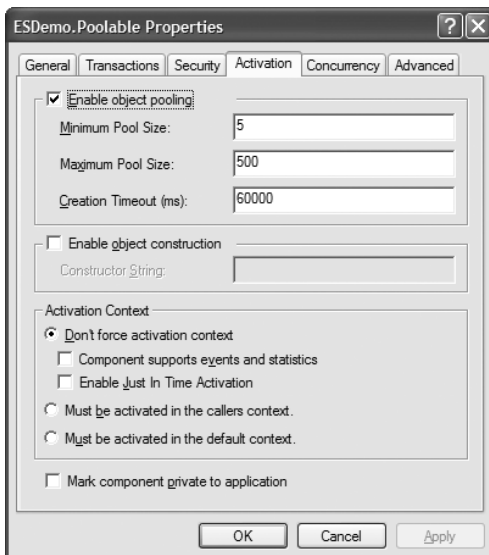


Figure 7-6. An object configured for pooling within Component Services

Transactions

COM+ can also manage transactions. Transactions can span methods, components, databases, and even servers running database products from different vendors. All code enlisted in the transaction gets to “vote” on the outcome of the transaction. It’s more like veto power though, because any vote of “no” causes the entire transaction to be rolled back.

Each class configured in COM+ sets an attribute determining its transactional behavior. This attribute is designed so that a transaction can be dynamically composed of many different components, in a way that may not be known when the autonomous components are designed. Table 7-4 is a summary of the options for transactional behavior.

Table 7-4. Options for Configuring the Transactional Behavior of COM+ Components

Transaction Option	Meaning In Life
Requires	COM+ will create a new transaction if none exists, or the object will be enlisted in the transaction of the object that created it.
Requires New	COM+ will create a brand new transaction. The object will not participate in the transaction of its creator, if one exists.
Supports	The object will be enlisted in the transactional support of the creating object (if it has any) or else will run without a transaction.
Does Not Support	This is the default. The object does not care about, and does not participate in, any transactions.
Disabled	This is like Does Not Support, but it requires no COM+ context. This choice is nearly equivalent to a nonconfigured component.

Transactions are always kicked off by a *root object*, which is a component flagged as either requiring a transaction or requiring a new transaction. This root object acts as the manager of the transaction, enlisting the appropriate set of autonomous methods to accomplish the transactional work. If any single method enlisted in the transaction votes “no,” the entire transaction is doomed to fail. All of the other objects enlisted in the transaction are called *secondary objects*. They need to be flagged with the Supports or Requires option. If they are flagged with Requires New, they become a new root object in a new transaction that will succeed or fail independently from the transaction that created it. This allows for a very flexible design of the transactions to reflect complex business processes (see Figure 7-7).

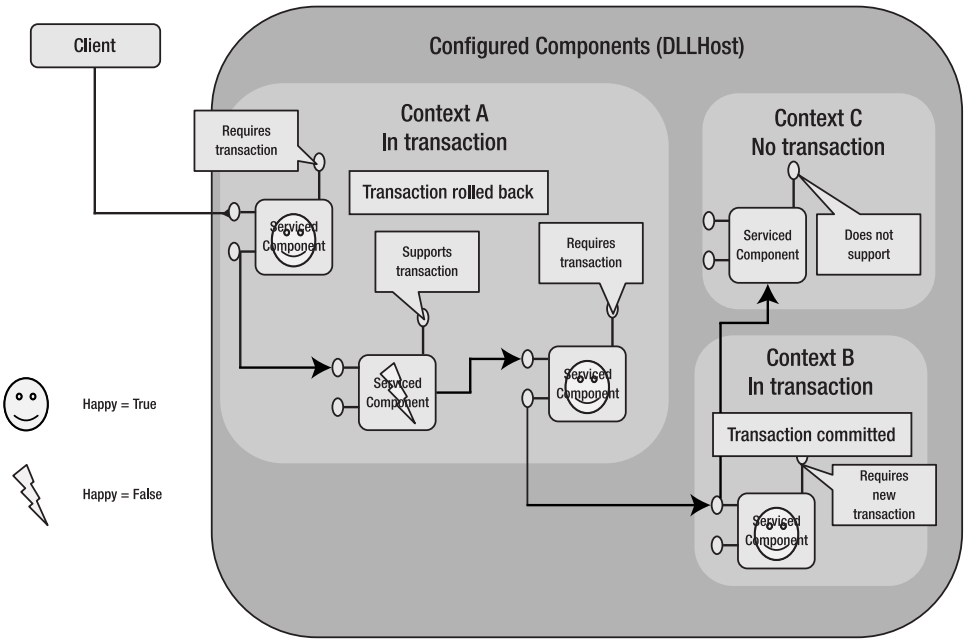


Figure 7-7. A complex set of transactions, where one transaction failed and another succeeded, while a third component did its work outside of the context of any transaction

There are a couple of options for a component to vote on the outcome of a transaction from code. One is declarative and the other imperative. The declarative method is very easy. A method can be flagged with the `AutoComplete` attribute, and as long as the method body does not throw an exception, the method's vote will be to commit the transaction. The method must be coded such that if something goes wrong, an exception is thrown. If an exception is thrown by the database, it must be left to ascend the call stack, or be trapped, wrapped, and rethrown. Any violation of business rules that results in the method being unable to complete its work must also be handled by throwing an exception. (You can find this code in `XActional.cs` in the `Serviced` project.)

```
[Transaction(TransactionOption.Required)]
public class CustomerService
{
    [AutoComplete(true)]
    public void IncreaseCreditLimit(
        int customerNum,
        double increaseAmount)
    {
        try
        {
            Customer cust = new Customer(customerNum);
            double max = cust.MaxAllowableCredit;
            double current = cust.CreditLimit;

            if (max < current + increaseAmount)
            {
                throw new Exception("Max Credit Limit Exceeded");
            }
            cust.CreditLimit += increaseAmount;
            cust.Save();
        }
        catch (Exception ex)
        {
            throw new Exception(
                "Attempt to increase limit failed", ex);
        }
    }
}
```

Notice the explicitly thrown exception captures the violation of a business rule. The `try/catch` block traps and wraps the explicitly thrown exception, or any other exception that bubbles up the call stack from your calls into the `Customer` object.

The other option is to explicitly vote on the transaction outcome from within the body of your method. In this case, you would omit the `AutoComplete` attribute, and use the `ContextUtil` properties to indicate the success or failure of the work that's been done.

```

[Transaction(TransactionOption.Required)]
public class CustomerService
{
    public void IncreaseCreditLimit(
        int customerNum,
        double increaseAmount)
    {
        try
        {
            Customer cust = new Customer(customerNum);
            double max = cust.MaxAllowableCredit;
            double current = cust.CreditLimit;

            if (max < current + increaseAmount)
            {
                ContextUtil.MyTransactionVote = TransactionVote.Abort;
            }
            else
            {
                cust.CreditLimit += increaseAmount;
                cust.Save();
                ContextUtil.MyTransactionVote = TransactionVote.Commit;
            }
        }
        catch (Exception ex)
        {
            ContextUtil.MyTransactionVote = TransactionVote.Abort;
            throw new Exception(
                "Attempt to increase limit failed", ex);
        }
    }
}

```

It's only necessary to use this option when you do not want to throw exceptions when a component is unable to finish its work. Generally, unless it interferes with a larger error-handling strategy, you should use the `AutoComplete` attribute, as this results in much cleaner code overall.

Transactions are managed under the hood of COM+ by the Distributed Transaction Coordinator, a separate Windows Service. This service must be running for COM+ transactions to work. It is an expensive resource, and you must make considerations for the overhead your application will incur when you decide to use it. MSDTC is not the only technology available to manage transactions; you should consider other less-expensive options before deciding to go into COM+. A few bars for entry into COM+ for transactional management exist.

- Your transaction spans data sources, especially if it spans different relational databases. For example, if your transaction is moving information from Microsoft SQL Server into an Oracle database, MSDTC is an excellent option for managing the transaction.
- Your application has complex requirements around transactional composition. In this case, your services are designed to do different, autonomous pieces of work. Transaction coordinators are written that call these different services to accomplish a specific business process. The number of ways these services can be combined is high, or the requirements change and evolve often, and you anticipate introducing new transaction coordinators as newer versions of the product are introduced.
- Your application has customizable functionality, such that an end user, power user, or administrator has a tool that can affect how services are combined to do transactional work.

As a corollary to these guidelines, if you can meet your transactional requirements with another resource manager, your solution will probably perform better. One option may be using transactions within SQL Server Transact-SQL (TSQL). This limits the transaction to a single command execution from your data access code. A stored procedure can call other stored procedures to enlist in the work of the transaction. Another option is to use ADO.NET transactions. These transactions are tied to a connection so that many commands can be executed against a single data store and enlisted in a single transaction. You can use this to dynamically generate SQL or to combine multiple stored procedure calls. These options are not nearly as flexible as COM+ transactions, but they are much better performers; thus, you should leverage them when they can do the job.

See Chapter 12 for information on the transactional infrastructure built into the .NET Framework 2.0, which gives you options for “upgrading” transactions dynamically; this way, only the resource managers needed get enrolled on an as-needed basis.

Queued Components

Queued Components (QC) provide a layer of abstraction between the COM+ developer and MSMQ. Configuring a component as queued gives you all of the benefits of message queuing, including asynchronous method invocation, without having to worry about the underlying details of preparing MSMQ messages and placing them in queues.

When a component is configured as queued, a call to the message prompts COM+ to prepare a MSMQ message and place it in a private queue. Another COM+ process acts as a listener to that queue, pulls the message out when it arrives, and invokes the method described by the message (see Figure 7-8).

QC provides a fire-and-forget model of service invocation. For this reason, methods that are configured as queued cannot have a return value. The caller does not wait for a return value; instead, the caller continues execution as soon as COM+ prepares the message and gets it into the queue. The actual work the method does occurs asynchronously with whatever code path the caller continues with after the method call.

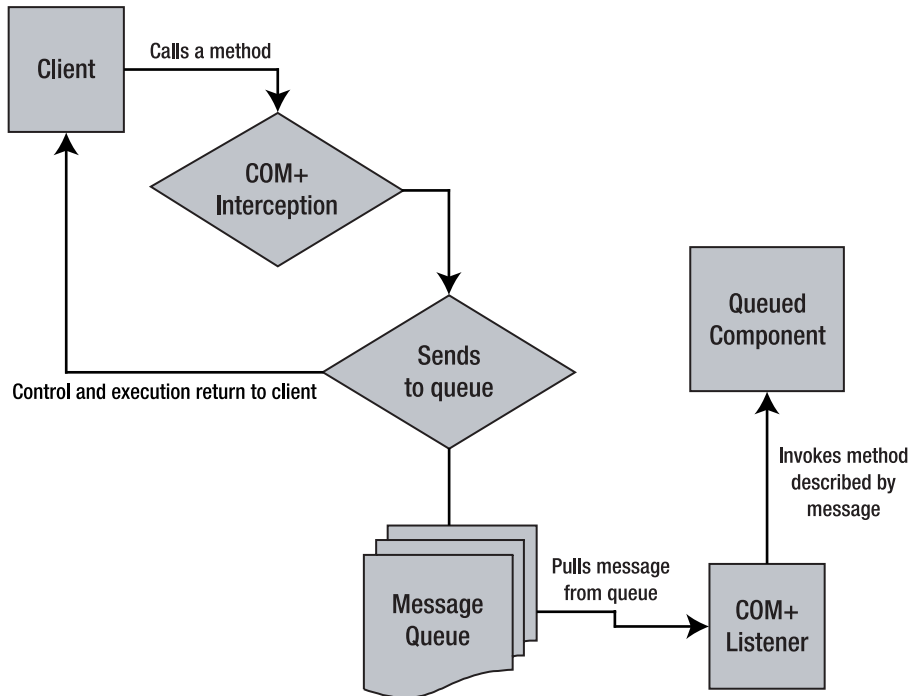


Figure 7-8. *The process of a method call to queued component*

Creating queued components is a little more involved than some of the other features of COM+. You'll need to do specific configurations on the server and on the client. You must add an assembly-level attribute to the assembly containing your components to queue. Finally, you must define an interface with all the methods you're planning to queue for a component, and then your Serviced Component must implement the interface.

When a client creates an instance of the component, it won't automatically call it using queuing. The good part about this is that the client has the flexibility to call the component in a queued manner or not. The downside is the need to write some specific code to leverage the queuing functionality. (You can find this queuing code in the Client and Server projects in the QCDemo directory of the Code07 solution.)

Let's start by taking a look at the assembly-level attribute that must be present.

```
[assembly: ApplicationQueuing(Enabled = true, QueueListenerEnabled = true)]
```

This sets up the application containing the component so that it can be queued. You can control the maximum number of listener threads with the attribute as well. The default is 16 per processor on the hosting machine.

Next you need to define an interface that you'll bind your queued messages to. An interface is necessary because you're using a COM-based technology, and all COM objects implement at least one interface. Queuing is, therefore, exposed on an interface-specific level of scope. So your Serviced Component must explicitly implement an interface, which your client will use to bind to and leverage the queued behavior.

You'll define an interface containing a single method. Remember that methods on this interface will not be able to return values; therefore, any method used with queuing must be declared as returning void.

```
public interface IQueueable
{
    void executeSQL(string sql);
}
```

Now you'll create a Serviced Component that implements this interface:

```
[InterfaceQueuing(Interface = "IQueueable")]
public class QCDemo : ServicedComponent, IQueueable
{
    public QCDemo() {}

    public void executeSQL(string sql)
    {
        try
        {
            SqlCommand cm = new SqlCommand(sql, new SqlConnection(ConnStr));
            cm.Connection.Open();
            cm.ExecuteNonQuery();
        }
        finally
        {
            cm.Connection.Close();
        }
    }
}
```

Notice you have decorated the type with the `InterfaceQueuing` attribute. With this attribute you're declaring your intent to use the named interface with in a queued manner. This will affect how the component gets configured (see Figure 7-9).

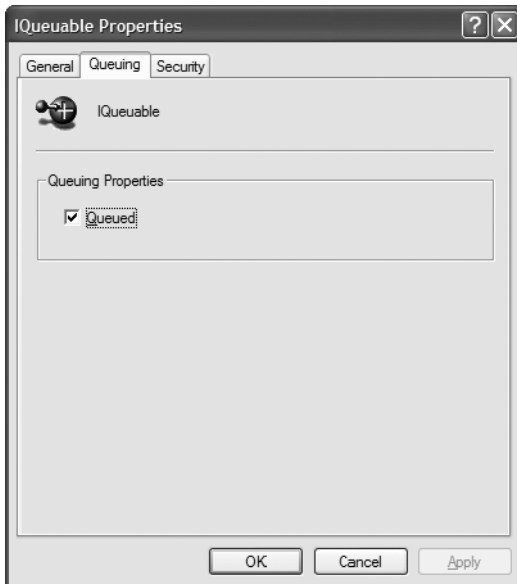


Figure 7-9. *An interface configured for queuing*

Your class is then declared as inheriting from `ServicedComponent` and implementing `IQueueing`. Clients will now have the choice of invoking this type via queuing or not. Here's a simple application to test your component:

```
static void Main(string[] args)
{
    string sql = "insert into jobs ( job_desc, min_lvl, max_lvl) "
                + "values ('Some job',10,250)";
    QCDEMO o = new QCDEMO();
    o.Dispose();
    Console.WriteLine("Component registered. Press enter to invoke");
    Console.ReadLine();
    IQueueable qable;
    try
    {
        qable = (IQueueable)Marshal.BindToMoniker
                ("queue:/new:Server.QCDEMO");

        for(int i = 0; i < 100; i++)
            qable.executeSQL(sql);
    }
    finally
    {
        Marshal.ReleaseComObject(qable);
    }
}
```

Configuration of a queued component can take a long time. Since you're relying on *lazy registration* (see the "Configuration" section a bit later in this chapter), your first block of code creates an instance of the type specifically for the purpose of registering it in COM+. You then declare an instance of the `IQueueable` interface and instantiate by using the `Marshal.BindToMoniker` method. This method lives in the `System.Runtime.InteropServices` namespace, which must be imported with a `using` statement. The string passed to it has the fully qualified name of the type built into the tail end of it, which is how the `BindToMoniker` method knows the proper type to create. The Serviced Component infrastructure takes care of the rest. Your code proceeds to call the `executeSQL` method 99 times. These method calls do not wait for the work of the insert statements to get done. Execution continues (and in our case, the application terminates) while the database work is picked up off the queue by the COM+ listener and executes asynchronously independently from our application. The last thing you do is explicitly destroy the COM object by calling `ReleaseComObject` (another static method on the `Marshal` type).

Role-Based Security

COM+ has its own infrastructure for enforcing role-based security. You can apply roles at the component, interface, or method level.

From Serviced Components, the .NET developer has two main tasks: creating COM+ roles and enforcing security at the appropriate level to make sure a caller is in the required role for the service it's attempting to call. Role creation is done with an assembly-level attribute:

```
[assembly: SecurityRole("Executive")]
[assembly: SecurityRole("Director")]
[assembly: SecurityRole("Manager")]
[assembly: SecurityRole("Grunt")]
```

These attributes result in the corresponding COM+ roles that are created (see Figure 7-10).

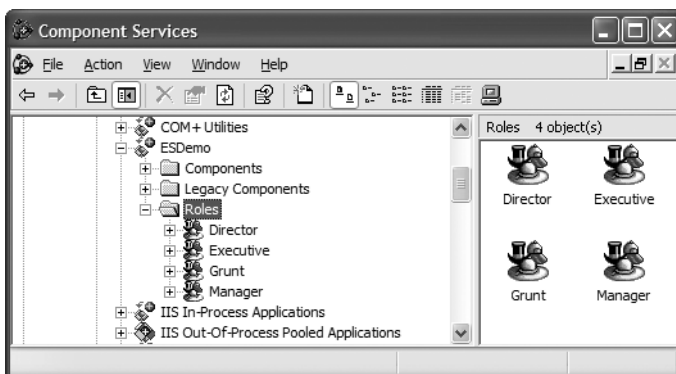


Figure 7-10. Roles created in COM+ with .NET assembly-level attributes

Role membership can now be enforced via either declarative aspects or imperative code. The declarative option is attractive, as it saves you from pushing complex conditional logic

into the code, and enables you to simply decorate your types with declarations of their security requirements. You use the `SecurityRole` attribute for this purpose as well. (This code is in the `Serviced` project of the `Code07` solution.)

```
[SecurityRole("Manager")]
public class RBDemo
{
    public RBDemo() {}

    public DataSet GetManagerData()
    {
        //implementation
        return new DataSet();
    }

    [SecurityRole("Executive")]
    public DataSet GetExecutiveData()
    {
        //implementation
        return new DataSet();
    }
}
```

In this class, callers to any method must be in the `Manager` role. You've further restrained access to the `GetExecutiveData` method, requiring that callers to that method are in the `Executive` role. You could also apply the attribute to an interface declaration.

Sometimes you need a finer grain of control over your role-based security implementation. For example, you may want to render a list of reports, and user roles determine access to the reports. In these cases, you'll need programmatic access to the roles information. This is exposed to use via the `SecurityCallContext` type. While this may at first appear to be similar to `ContextUtil`, security context is different, so using `SecurityCallContext` is required in this case.

```
public void GetReportData(DataSet reportCriteriaData)
{
    // Get the current security call context
    SecurityCallContext callCtx = SecurityCallContext.CurrentCall;

    // Verify role based security is enabled (optional)
    if (callCtx.IsSecurityEnabled)
    {
        // Only allow managers to generate reports
        if (callCtx.IsCallerInRole("Manager"))
        {
            // proceed with report generation
        }
        else
        {

```

```

    // security error
  }
}
}

```

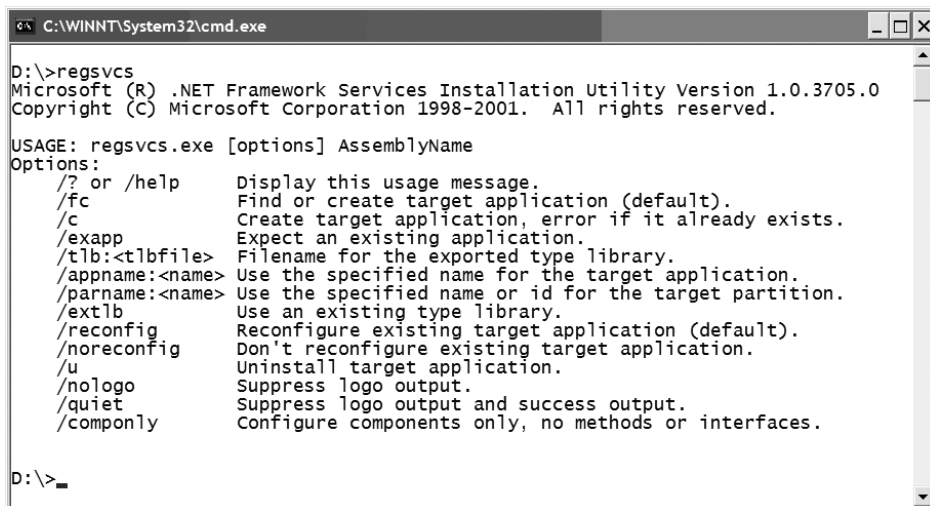
Using imperative coding enables you to introduce different flow-of-control scenarios into your code based on the role of your calling users.

Configuration

Once you get your classes inheriting from the Serviced Component base class written and compiled, you'll need to configure them within the Component Services environment. You can be lazy or proactive about this.

The lazy approach enables the runtime to do this the first time someone creates an instance of the type. Seriously, it's called *lazy loading*. The nice thing about this approach is that there's no additional setup or installation step that needs to occur. The first time the component is requested, the runtime makes sure the COM+ application exists and checks all of the other assembly-level attributes (such as role declarations) to make sure they're present as well. Anything that doesn't exist will be created. If the application already exists, it will be shut down and the changes applied. Components and their interfaces then get registered and configured with COM+. The best reason to use this type of registration is right in its name: laziness. This approach should be avoided if at all possible.

The downside to lazy registration is that there's a significant performance hit on the first request to your application. Especially if your app is using queued components, this delay can last as long as a few seconds. The other, probably more serious, drawback is that the user running the process has to be an administrator to have the appropriate permissions to do this. So when you're running Serviced Components from ASP.NET, the user running the ASP.NET Framework must be configured as an administrator. This is usually a show stopper for folks, and leads them to your second option: registering the components yourself using a command line tool (seen with its options displayed in Figure 7-11).



```

C:\WINNT\System32\cmd.exe

D:\>regsvcs
Microsoft (R) .NET Framework Services Installation Utility Version 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

USAGE: regsvcs.exe [options] AssemblyName
Options:
  /? or /help    Display this usage message.
  /fc            Find or create target application (default).
  /c            Create target application, error if it already exists.
  /exapp        Expect an existing application.
  /tlb:<tlbfile> Filename for the exported type library.
  /appname:<name> Use the specified name for the target application.
  /parname:<name> Use the specified name or id for the target partition.
  /extlb        Use an existing type library.
  /reconfig     Reconfigure existing target application (default).
  /noreconfig   Don't reconfigure existing target application.
  /u            Uninstall target application.
  /nologo       Suppress logo output.
  /quiet        Suppress logo output and success output.
  /componly     Configure components only, no methods or interfaces.

D:\>

```

Figure 7-11. The help screen for the `regsvcs` (register services) command line utility

To use this utility, you simply feed the tool the name of the assembly containing classes you need to configure with the /c switch for “configure.” You can also use /fc if there’s a possibility that the application already exists, and you just want to add your types to it.

After registration with COM+, the .NET attributes applied to your types will determine the default configuration of the components. You (or an administrator) can always go in and use the MMC snap-in for Component Services to change and further refine them.

Some Practices Worth Observing

There are a lot of details to keep in mind when you’re using component services. While a comprehensive discussion of the details of contexts, COM Interop, and an exhaustive list of best practices is out of scope for the summary we present in this chapter, we advise you keep some simple things in mind as you move into development in this space.

Component Design

First and foremost, create your applications with a stateless design. Don’t use field-level information that creates a dependency across method calls. Have each method be truly autonomous and isolated. If and when you need to maintain state, do so in the database, and pass a session ID to the user that he can use to later retain the state information. Don’t initialize connections in the constructor of your type to use it from the different methods of your type. Don’t assume your user is going to be responsible in the use of your type by creating late, and destroying early; use JITA-enabled components instead.

To more explicitly separate the interface of your component from the implementation, you should always create an interface for your Serviced Component to implement.

An interface will be created under the hood to represent your class. By default, it will have the name *_ClassName*. You might as well create your own interfaces, and have your Serviced Component implement them. This means the interfaces listed within COM+ will be known types you’ve intentionally created; this will also ease the deployment of metadata if you’re using COM+ in a distributed architecture.

Do not use static methods on types that inherit from *ServicedComponent*. These are not designed to work within COM+.

Also, always use the *ContextUtil* and *SecurityCallContext* to get to the underlying COM+ functionality. Do not call directly into the native COM+ libraries, as the behavior here will be volatile or perhaps merely unpredictable at best. The Enterprise Services assembly has been created with a layer of interoperability specifically designed and optimized for Serviced Components. Use it.

Security Contexts

You generally want to avoid using impersonation. This is true for Component Services, but is more generally true for the middle tier. Impersonation means that a unique user is used to execute the code for each session of the application. There is a security context involved here, and so pooled objects will not be shared across users. This can largely defeat the whole purpose of pooling in the first place. It can actually make things worse, as a pool per user may create far more instances than are actually needed.

Object Lifetime

As a consumer of Serviced Components, always call `Dispose` on an instance when you're finished with it. The easiest way to do this is to use the `using` statement. This guarantees `Dispose` gets called on the type, regardless of your error-handling semantics.

```
private static void ExPool()
{
    string[] tables =
        { "authors", "employee", "titles", "publishers", "sales" };
    Random r = new Random();
    using (Poolable p = new Poolable())
    {
        for (int i = 1; i < 10; i++)
        {
            string s = string.Format(
                "select * from {0}", tables[r.Next(tables.Length - 1)]);
            DataSet ds = p.GetSomeData(s);
        }
    }
    //Dispose called automatically when 'using' goes out of scope
}
```

If you're creating Serviced Components you can control instance lifetimes by using JITA (see previous section on JITA). Instead of explicitly calling `DeactivateOnReturn` from each method implementation, you also have the option of simply flagging your class with the `AutoComplete` attribute. This attribute will guarantee instances are disposed of after a method completes. If your type is not JITA-enabled, the `AutoComplete` attribute will be ignored.

Configuration and Deployment

Only use the attributes from the Enterprise Services assembly that you really need. Each of the features we've looked at is configured with attributes. Each incurs some overhead. Leaving a component unconfigured for a given feature means no overhead will be incurred for leveraging that feature. Be conscious and deliberate about which of these you need.

For production deployments, you should always use `Regsvcs.exe` to register your Serviced Components in COM+. Lazy registration is a convenient feature, but one that should only be enjoyed during development.

`Regsvcs.exe` will automatically put your Serviced Components into the Global Assembly Cache (GAC). While this is convenient, any assemblies your Serviced Component is dependant upon will not enjoy the same convenience. This could cause a problem at runtime. For this reason, your deployment should also explicitly register your components and their dependencies in the GAC.

Summary

Component Services exposes a rich set of features that you can consume from .NET by having your types inherit from `ServiceComponent` and decorating your types with attributes that determine how they'll be configured when they're registered with COM+.

Because this is still a COM-based technology, a layer of interop is used; therefore, you must be sure the benefits of the features you're leveraging outweigh the performance hit you'll incur by using the environment. Many of the features of COM+, when needed, will be worth this performance hit.

Once your components are created and configured within COM+, calling processes will need to get to them. This may happen in process from IIS, it may happen via DCOM from a Windows Forms application, or it may happen via Web Services. In the next chapter, we'll examine some of the different options that are available for invoking Service Components from within different application architectures.