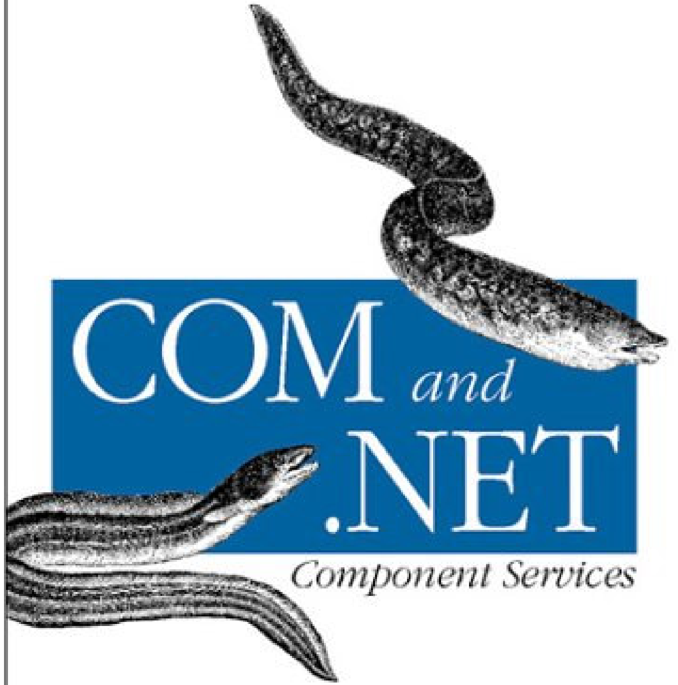
Migrating from COM+ to .NET



COM *and* .NET *Component Services*

O'REILLY®

Juval Löwy

COM and .NET Component Services

Dedication

To my wife, Dana

Foreword

I first ran into COM+ back in 1996. In those days, I was working as a Common Object Request Broker Architecture (CORBA) consultant and was fresh out of IBM, where I had been heavily involved in IBM's original CORBA implementation.

CORBA was the first of the architectures that we might describe today as Distributed Component architectures, which set the stage for both COM/DCOM in the Microsoft space and RMI/IIOP in the Java space.

Back then, I was interested in a particularly knotty problem related to distributed component architectures. Systems built with such architectures had a characteristic performance pattern. They could handle large numbers of transactions, as long as those transactions originated from a small number of clients. So, for example, 5,000 transactions per minute divided between 5 clients worked fine. But when those same 5,000 transactions per minute were split among 1,000 clients, each processing 5 transactions per minute, the systems choked.

This was odd, I thought. Why should 5 clients, each processing 1,000 transactions per minute, be fundamentally different than 1,000 clients, each processing 5 transactions per minute? What is the difference between the first 5,000 transactions per minute and the second?

Distributed component architectures, as they existed in 1996, dictated a one-to-one relationship between clients and component instances. The business logic of such architectures is in the component instances. And it is the business logic that makes transactional requests of transactional resources, such as the database. In order to make transactional requests, the component instances require expensive resources, such as database connections. We run out of steam (i.e., transactional throughput) when one of two things happen: we overload the system with transactional requests or we run out of resources (e.g., database connections).

Clearly, going from 5 clients, each making 1,000 transactional requests per minute, to 1,000 clients, each making 5 transactional requests per minute, has no overall impact on the transactional throughput. Therefore, the reason why our distributed component systems must be dying is that we are running out of resources.

So the answer to getting lots of clients on a distributed component architecture is not going to come from increased capability of the back-end transactional resources (e.g., databases). It will have to come from something else-something that allows resource sharing.

This, then, is the problem I worked on back in 1996. How do you

get several clients to share resources in a distributed component architecture?
The solution to this problem came from an unexpected source. I was asked to write a book on COM and DCOM. I knew very little about COM and DCOM back then. As I looked over the COM/DCOM white papers on the Microsoft web site, I quickly recognized it as a typical distributed component architecture and predicted the same throughput problems I had seen in other distributed component systems.
As I browsed through the white papers, I noticed an obscure beta product called Microsoft Transaction Server (MTS). At first, I dismissed MTS as an API used to manage distributed transactions. But as I read more about MTS, I realized that it had little to do with transactions. Instead, it attacked a much more interesting problem: how to share resources among clients. In a nutshell, MTS addressed the very problem that had so vexed the existing distributed component systems-how to support a large number of low-transaction generating clients!
I did eventually write that book, as well as many articles on the importance of the ideas introduced by MTS. Many of these articles appeared in my ObjectWatch newsletter (available at www.objectwatch.com), a newsletter that has, over time, become influential in its space.
Back in 1996, I predicted that MTS would be a historically important product-one that would redefine approaches to scalability in distributed component systems. In fact, that prediction has come true. Today, every infrastructure designed to support high scalability in distributed component systems is based directly on the ideas, algorithms, and principals first introduced by MTS in 1996. Enterprise JavaBeans, for example, the Java scalability infrastructure, is almost a direct copy of MTS.
But what does this have to do with COM+, you may ask. It turns out that COM+ and MTS are one and the same. Microsoft, never known for its marketing savvy, decided to wait until customers finally got used to the name MTS (itself a misleading name), and then it pulled a fast one-it switched the name! And not just any name, but one that would be as confusing as possible! So they renamed MTS as COM+. Naturally, customers assumed that COM+ was the next release of COM. In fact, COM+ was the next release of MTS.
Now Microsoft has announced .NET. Once again, the brilliant Microsoft marketing organization has left many customers confused. Is COM+ now dead? Far from it—.NET is a series of interesting new features, none of which replace COM+. COM+ is still the scalable infrastructure that supports resource sharing and deals with the myriad of issues (such as security and transaction boundary

management) that are so closely related to resource sharing and so crucial to distributed applications.
So whether you are rushing into Microsoft's new .NET technology platform or taking a wait and see attitude, if you need to put a lot of clients around your system, you need to understand COM+. Therefore, this book is very timely. COM+ is going to be with us for a long time. Its name may change again, just to confuse the innocent; but the ideas, algorithms, and principals will not. COM+, under whatever name, is here to stay!
Roger Sessions,
CEO, ObjectWatch, Inc.
Publisher, *ObjectWatch* newsletter (www.objectwatch.com)
Author, *COM+ and the Battle for the Middle Tier*
Austin, Texas

Preface

This book discusses COM+ component services. Each service is covered in its own chapter, and each chapter discusses a similar range of issues: the problem the service addresses, possible solutions to that problem, an in-depth description of the COM+ solution, tradeoffs, design, and implementation guidelines, tips, and known pitfalls. I have tried to provide useful design information and lessons I learned while applying COM+. I also describe COM+ helper classes and utilities I developed that will enhance your productivity significantly. (The COM+ Events helper objects and the COM+ Logbook are prime examples.) This book focuses on the "how to"—that is, it provides practical information. You should read the chapters in order, since most chapters rely on information discussed in the preceding chapters. The book also aims to explain COM+ step by step. A software engineer already familiar with COM who wants to know what COM+ is and how to use it can read this book and start developing COM+ applications immediately.

Scope of This Book

Here is a brief summary of the chapters and appendixes in this book:

- Chapter 1 introduces the Component Services Explorer and basic COM+ terminology. This chapter deliberately holds your hand as you develop your first "Hello World" COM+ component. Subsequent chapters do much less handholding and assume you are familiar with the COM+ environment. If you already have experience with basic COM+ development, feel free to skip this chapter.
- Chapter 2 demystifies the COM+ context by presenting it as the key mechanism for providing component services using call interception. Generally, you need not be concerned with contexts at all. However, the COM+ context underlies the way COM+ services are implemented.
- Chapter 3 describes two scalability-enabling mechanisms that COM+ provides for a modern enterprise application: object pooling and Just-in-Time Activation (JITA). The discussion of instance management, and especially JITA, is independent of transactions. Early COM+ documentation and books tended to couple instance management and transactions. However, I found that not only can you use instance management independently of transactions, but it is easier to explain it that

way. Besides explaining how to best use object pooling and JITA, Chapter 3 describes other activation and instance management COM+ services such as the constructor string.

- Chapter 4 explains the difficult, yet common, problems that transactions address, and provides you with a distilled overview of transaction processing and the transaction programming model. The difficult part of writing this chapter was finding a way to convey the right amount of transaction processing theory. I want to help you understand and accept the resulting programming model, but not bury you in the details of theory and COM+ plumbing. This chapter focuses on COM+ transaction architecture and the resulting design considerations you have to be aware of.
- Chapter 5 first explains the need in the component world for a concurrency model and the limitations of the classic COM solution. It then describes how the COM+ solution, activities, improves deficiencies of apartments.
- Chapter 6 shows how to access component and application configuration information programmatically using the COM+ Catalog interfaces and objects. Programmatic access is required when using some advanced COM+ services and to automate setup and development tasks. This chapter provides you with comprehensive catalog structure diagrams, plenty of sample code, and a handy utility.
- Chapter 7 explains how to secure a modern application using the rich and powerful (yet easy to use) security infrastructure provided by COM+. This chapter defines basic security concepts and shows you how to design security into your application from the ground up. You can design this security by using COM+ declarative security via the Component Services Explorer and by using advanced programmatic security.
- Chapter 8 explains what COM+ queued components are and how to use them to develop asynchronous, potentially disconnected applications and components. In addition to showing you how to configure queued components, this chapter addresses required changes to the programming model. If you have ever had to develop an asynchronous method invocation option for your components, you will love COM+ queued components.
- Chapter 9 covers COM+ loosely coupled events, why there is a need for such a service, and how the service ties into other COM+ services described in earlier chapters (such as transactions, security, and queued components). Many people consider COM+ events their favorite service. If you have had to confront COM connection points, you will appreciate COM+ Events.

- Chapter 10 shows how .NET components can take advantage of the component services described in the previous chapters. If you are not familiar with .NET, I suggest you read Appendix C first—it contains an introduction to .NET and C#. Chapter 10 repeats in C# many of the C++ or VB 6.0 code samples found in earlier chapters, showing you how to implement them in .NET.
- Appendix A helps you develop a useful and important utility—a flight recorder that logs method calls, errors, and events in your application. Logging is an essential part of every application and is especially important in an enterprise environment. The logbook is also an excellent example of the synergies arrived at by combining multiple COM+ services. It is also a good representation of the design approaches you may consider when combining services.
- Appendix B describes the changes, improvements, and enhancements introduced to COM+ in the next release of Windows, Windows XP. Instead of writing the book as if Windows XP were available now (as of this writing it is only in beta), I chose to write the book for the developer who has to deliver applications today, using Windows 2000. When you start using Windows XP, all you need to do is read Appendix B—it contains the additional information you need.
- Appendix C describes the essential elements of the .NET framework, such as the runtime, assemblies, and how to develop .NET components. The appendix allows a reader who is not familiar with .NET to follow Chapter 10.

Some Assumptions About the Reader

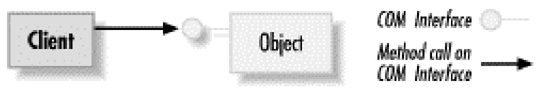
I assume that you are an experienced COM developer who feels comfortable with COM basics such as interfaces, CoClasses, and apartments. This book is about COM+ component services, not the component technology used to develop a COM/DCOM or .NET component. You can still read the book without this experience, but you will benefit more by having COM under your belt. I assume you develop your components mostly in C++ and ATL and that you write occasional, simple client code in Visual Basic. I also use trivial C# in Chapter 10 to demonstrate how .NET takes advantage of COM+ services, but you don't need to know C# to read that chapter. A .NET developer should also find this book useful: read and understand the services in Chapter 1 through Chapter 9, and then use Chapter 10 as a reference guide for the syntax of .NET attributes.

Definitions and Text Conventions

The following definitions and conventions apply throughout this book:

- A component is an implementation of a set of interfaces. A component is what you mark in your IDL file (or type library) with CoClass or a class in C#.
- An object is an instance of a component. You can create objects by calling `CoCreateInstance( )` in C++, specifying the class ID (the type) of the object you want to create. If you use Visual Basic 6.0, you can create objects using `new` or `CreateObject( )`. A C# client uses `new` to create a new instance of a component.
- I use the following terms in the book: CoCreating refers to calling `CoCreateInstance()` in C++, or `new` or `CreateObject( )` in Visual Basic. Querying an object for an interface refers to calling `IUnknown::QueryInterface( )` on the object. Releasing an object refers to calling `IUnknown::Release( )` on the object.
- The graphical notations in Figure P-1 are used in almost every design diagram in the book. The "lollipop" denotes an interface, and a method call on an interface is represented by an arrow beginning with a full circle.

Figure P-1. Interface and method call graphical notations



- Error handling in the code samples is rudimentary. The code samples serve to demonstrate a design or a technical point, and cluttering them with too much error handing would miss the point. In a production environment, you should verify the returned HRESULT of every COM call, catch and handle exceptions in C#, and assert every assumption.

I use the following font conventions in this book:

- *Italic* is used for new terms, citations, online links, filenames, directories, and pathnames.
- `Constant width` is used to indicate command-line computer output and code examples, as well as classes, constants, functions, interfaces, methods, variables, and flow-controlled statements.

- **`Constant-width bold`** is used for code emphasis and user input.
- *`Constant-width italic`* is used to indicate replaceable elements in code statements.

	This icon indicates a note or tip.
	This icon indicates a warning.

Other COM+ Books and References

This book describes how to use COM+ component services in your application. It focuses on how to apply the technology, how to avoid specific pitfalls, and design guidelines. If you want to know more about COM+ in general and the nature of component technology, I recommend the following two books that helped me a great deal in my attempt to grasp COM+.

COM+ and the Battle for the Middle Tier by Roger Sessions (John Wiley & Sons, 2000) is hands down the best "why" COM+ book. It explains in detail, with excellent examples and in plain language, the need for software components and component services. For example, instead of the page or two this book includes on the motivation for using transactions, Sessions devotes two fascinating chapters to the topic. The book goes on to compare existing component technologies (such as COM, CORBA, and Java) and their corresponding suites of component services. It also contains a few case studies from real-life systems that use COM+. Roger Sessions also has a unique way of eloquently naming things—providing the most appropriate term, which is often not the name Microsoft uses. Whenever it makes sense, this book uses Sessions' terminology, such as "instance management" instead of the Microsoft term "activation."

Understanding COM+ by David S. Platt (Microsoft Press, 1999) is probably the best "what" COM+ book. The book describes the services available by COM+ and provides sidebar summaries for the busy reader. It is one of the first COM+ books, and Platt worked on it closely with the COM+ team.

I also used the MSDN Library extensively, especially the "Component Services" section, while writing this book. Although the information in this library tends to be terse, the overall structure is good. Use this book to learn how to apply COM+ productively and effectively, and use the MSDN Library as a reference for technical details and a source for additional information.

How to Contact Us

We have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). Please address comments and questions concerning this book to the publisher:
O'Reilly & Associates, Inc.
101 Morris Street
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)
The web site for the book lists examples, errata, and plans for future editions. You can access this page at:
http://www.oreilly.com/catalog/comdotnetsvs
To ask technical questions or comment on this book, send email to:
bookquestions@oreilly.com
Or to me directly:
juval.lowy@componentware.net
For more information about our books, conferences, software, resource centers, and the O'Reilly Network, see our web site:
http://www.oreilly.com

Acknowledgments

A book is by no means the product of just the author's work. It is the result of many events and individuals, like links in a chain. I cannot possibly name everyone, ranging from my parents to my friends. I am especially grateful for my two friends and colleagues, Marcus Pelletier and Chris W. Rea. Marcus worked with me on large COM+ projects, and together we confronted the unknown. Marcus's thoroughness and technical expertise is a model for every programmer. Chris's comments and insight into a reader's mind have contributed greatly to this book's accuracy, integrity, and flow. I wish to thank Yasser Shohoud for verifying my approach to transaction processing and sharing with me his own, Richard Grimes for reviewing the book, and Roger Sessions for writing the Foreword. Thanks also to Johnny Blumenstock for providing me with a place to write. Finally, this book would not be possible without my

wife, Dana, whose constant support and encouragement made this book a reality. Thank you, Dana.

Chapter 1. COM+ Component Services

By now, most developers of large-scale enterprise applications are convinced of the benefits of component-oriented development. They have discovered that by breaking a large system down into smaller units, they can write code that is easier to reuse on other projects, easier to distribute across multiple computers, and easier to maintain. As long as these components adhere to a binary standard that defines how they communicate with one another, they can be invoked as needed at runtime and discarded when they have finished their work. This type of application is also particularly suited to the Web, where clients request services of remote applications and then, once satisfied, move on to other tasks.

For nearly a decade, the Microsoft Component Object Model (COM) has been the standard for components that run on Windows machines, including Windows 9x and Me clients and Windows NT and 2000 servers. The COM model is well documented by the Microsoft Component Object Model Specification. Tools such as Visual C++ and Visual Basic make it easy to create COM components, and scores of books, training classes, and articles are available to teach programmers how to use them. Many features of the Windows operating system are now implemented as COM components, and many companies have invested heavily in COM-based systems of their own.

In July 2000, Microsoft announced a radically new component model as part of its .NET development platform, suddenly calling into question the viability of existing COM applications. .NET components bear little resemblance to legacy COM components and are not backwards compatible. They can be made to interoperate with COM components but do not do so naturally.

When it comes to the services and tools programmers use to build enterprise-scale .NET applications, however, one facility continues to provide the necessary runtime infrastructure and services: COM+ component services. These services have been available on Windows 2000 since its release, but they will gain greater importance in the months ahead. As it turns out, they offer a bridge between traditional COM and .NET applications, making your understanding and mastery of them as important now as it has ever been.

In this chapter, we provide a quick overview of the COM+ suite of component services and then introduce you to the Component Services Explorer, your primary tool for building and managing both COM and .NET enterprise applications. You will also create, debug, and deploy a simple COM+ "Hello World" application, using a traditional COM component and learning about COM+ application types and configured components as you do so.

1.1 COM+ Component Services

Components need runtime services to work. The original COM runtime supported components located on the same machine, typically a desktop PC. As the focus of Windows development shifted from standalone PCs to networked systems, Microsoft found it necessary to add additional services (see The Evolution of COM+ Services). First, they added support for *distributed applications*, or applications whose components are located on more than one machine (sometimes referred to as "COM on a wire"). Later, Microsoft added new services to support *enterprise applications*, whose complexity and scale placed new demands on the resources of a system and required an entirely new level of support. These trends were only exacerbated by the move to web-based applications aimed at huge numbers of customers connected over the public Internet.

Collectively, the services that support COM and .NET component-based applications are known as the *COM+ component services*, or simply as COM+.

The Evolution of COM+ Services

COM solved a number of problems facing early component developers by providing a binary standard for components, defining a communication interface, and providing a way to link components dynamically. COM freed developers from having to deal with "plumbing" and connectivity issues, allowing them to concentrate on designing components.

By the mid-1990s, however, it was clear that Windows developers needed additional services to support distributed and transaction-oriented applications. Distributed COM (DCOM) was released in 1995, a specification and service used to distribute components across different machines and invoke them remotely. Then, Microsoft released the Microsoft Transaction Server (MTS) in 1998, which provided component developers with new services for transaction management, declarative role-based security, instance activation management, component deployment and installation, and an administration tool for managing component configurations.

There was more to MTS than just new services. MTS represented a programming model in which the component developer simply declared (using the MTS administrative tool) which services a component required, and left it to MTS to provide an appropriate runtime environment. Developers could now spend even less effort on low-level service plumbing (such as interacting with transaction processing monitors or managing the life cycle of an object), and more on the business logic the customer paid for. Yet, MTS had its limitations. Foremost was the fact that MTS was built on top of conventional COM/DCOM. The underlying operating system and COM itself were unaware that MTS even existed. MTS resorted to esoteric hacks and kludges to provide its services, and MTS could not provide its services to every COM object (multithreaded apartment objects were excluded). Some services, such as object pooling, were either not possible or unavailable.

The development of a new version of the Windows NT operating system (initially called NT 5.0 and later renamed Windows 2000), gave Microsoft an opportunity to correct the deficiencies of MTS and DCOM by fusing them into a new comprehensive component services suite. Microsoft added yet more services, including object pooling, queued components, and events, and made the package a part of the core Windows operating system. The new suite of services was named COM+ 1.0, the subject of this book. The next version of COM+, COM+ 1.5, is scheduled for release with Windows XP in Q4 2001 and is described in Appendix B.

The COM+ acronym is an overloaded and often misused term. Today it is used informally to refer to both the latest version of the COM component specification and the component services available on the latest versions of Windows. In this book, we use the term COM+ to refer to the COM+ component services. When we speak of COM+ components, we refer to COM components configured to run under those services. However, as you will see, a COM+ application may consist of either COM or .NET components (see COM+: The Migration Path to .NET).

Here is a quick summary of the most important services provided by COM+:

Administration

Tools and services that enable programmers and administrators to configure and manage components and component-based applications. The most important tool is the Microsoft Management Console Component Services Explorer. COM+ also provides a standard location, the COM+ Catalog, for storing configuration information. The Component Services Explorer is explained in the following section. The COM+ Catalog is described in Chapter 6.

Just-in-Time Activation (JITA)

Services that instantiate components when they are called and discard them when their work is done. JITA is explained in Chapter 3.

Object pooling

Services that allow instances of frequently used, but expensive, resources, such as database connections, to be maintained in a pool for use by numerous clients. Object pooling can improve the performance and responsiveness of a distributed application dramatically. It is explained in Chapter 3.

Transactions

Services that allow operations carried out by distributed components and resources such as databases to be treated as a single operation. Transaction management is a requirement of most commercial systems. COM+ Transaction services are discussed in Chapter 4.

Synchronization

Services for controlling concurrent access to objects. These services are explained in Chapter 5.

Security

Services for authenticating clients and controlling access to an application. COM+ supports role-based security, which is explained in Chapter 7.

Queued components

Services that allow components to communicate through asynchronous messaging, a feature that makes possible loosely coupled applications or even disconnected applications. Queued components are discussed in Chapter 8.

Events

Services that allow components to inform one another of significant events, such as changes in data or system state. COM+ supports a publish-subscribe model of event notification, which is described in Chapter 9.

To summarize, COM+ is about component services and has almost nothing to do with the way a component is developed. The .NET framework allows you to develop binary components more easily than does COM, but it continues to rely on component services available through COM+. The manner in which .NET and COM components are configured to use these services, however, is not the same. Currently, most Windows enterprise developers are developing applications based on the existing COM standard using Visual Basic 6 and Visual C++ 6 with ATL. For this reason, this book uses COM examples to demonstrate COM+. However, these same services are available to .NET components as well. Chapter 10 shows you how to use them.

COM+ 1.0 is an integral part of Windows 2000 and requires no special installation. Some COM+ features are available only when both the client and server are running on Windows 2000 machines, but COM+ clients can usually run on Windows 9.x and Windows NT machines as well.

COM+: The Migration Path to .NET

.NET is Microsoft's next-generation component technology and application development platform. (For a quick overview of the .NET platform, see Appendix C.) However, adopting a radically new technology such as .NET is never an easy endeavor for companies and developers. Most have made a considerable investment in an existing, often COM-based, code base and the developer skills needed to maintain it. Unless companies have a compelling reason to move to .NET or a reasonable migration path, they postpone or avoid making the change.

However, because COM and .NET components can coexist in the same COM+ application, companies can continue to build COM components today, adding .NET serviced components to their applications at a later time when the advantages of doing so are more compelling. This is a migration strategy worth your consideration.

When Windows XP is released in Q4 2001, it will include a new version of COM+ component services, COM+ 1.5. This new version improves COM+ 1.0 usability and addresses some of the pitfalls of using COM+ 1.0 on Windows 2000, as described in this book. COM+ 1.5 also adds new features to existing services and lays the foundation for integration with .NET web services. Appendix B summarizes the forthcoming changes.

1.2 The Component Services Explorer

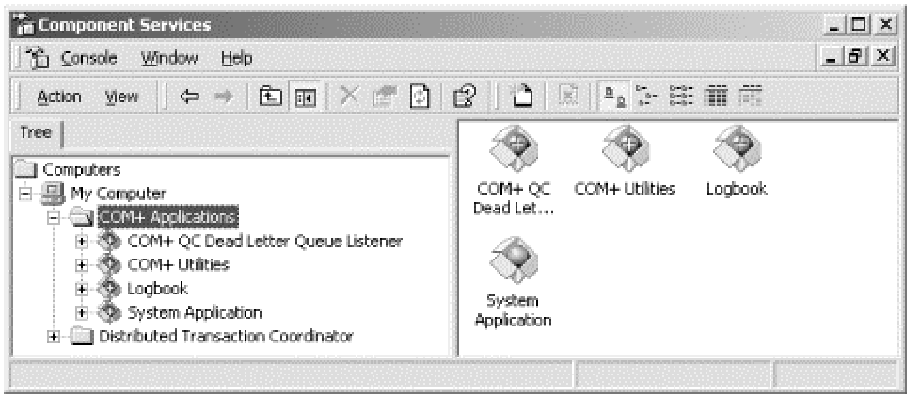
COM+ components and applications are managed through the *Component Services Explorer* (formerly known as the *COM+ Explorer*).The Component Services Explorer is a Microsoft Management Console snap-in and is available on every Windows 2000 machine.

To fire up the Component Services Explorer, go to the Start menu and select Settings →Control Panel. When the Control Panel window appears, select the Administrative Tools directory and then select the Component Services application.

The first thing you should do after locating the Component Services Explorer is create a shortcut to it on your desktop. As a developer, you need easy access to the Component Services Explorer, your main gateway into COM+ (see Figure 1-1). You can use the Component Services Explorer to create and configure COM+

applications, import and configure COM or .NET components, export and deploy your applications, and administer your local machine. You can even administer COM+ on other machines on the network, provided you have administrative privileges on those machines.
A *COM+ application* is a logical group of COM+ components. Components usually share an application if they depend on one another to accomplish their tasks and when all the components require the same application level configuration, as with security or activation policy. Components in the same application are often developed by the same team, and are meant to be deployed together.
You can see all the COM+ applications installed on your machine by opening the Component Services Explorer and expanding the *Computers* folder in the Tree window: Computers →My Computer →COM+ Applications. Every icon in the *COM+ Applications* folder represents a COM+ application. Each COM+ application contains COM+ components. Components must be explicitly imported into the Component Services Explorer to take advantage of COM+ services.
The Component Services Explorer offers a hierarchical approach to managing COM+ services and configurations: a computer contains *applications,* and an application contains *components.* A component has *interfaces,* and an interface has *methods.* Each item in the hierarchy has its own configurable properties. Note that the hierarchy allows you to view the parameters of any method listed in the hierarchy.

Figure 1-1. The Component Services Explorer



1.3 Hello COM+

The best way to become acquainted with the Component Services Explorer and basic COM+ terminology is to do a trivial example. This section walks you through the COM+ equivalent of the

canonical "Hello World" program. You will build a COM+ application containing a COM component that displays a message box saying "Hello COM+".
When developing your "Hello COM+" application, follow these steps:

1. Create a classic COM component. All COM+ components start their life as classic COM components, developed with such tools as ATL, MFC, or Visual Basic 6.0.
2. Create a new COM+ application to host the component.
3. Add the component to the application.
4. Write a client and test the component.

The rest of this chapter uses this "Hello COM+" example to demonstrate various COM+ features and capabilities. The example is also available as part of the source files provided with this book (see the Preface for information on how to access these files).

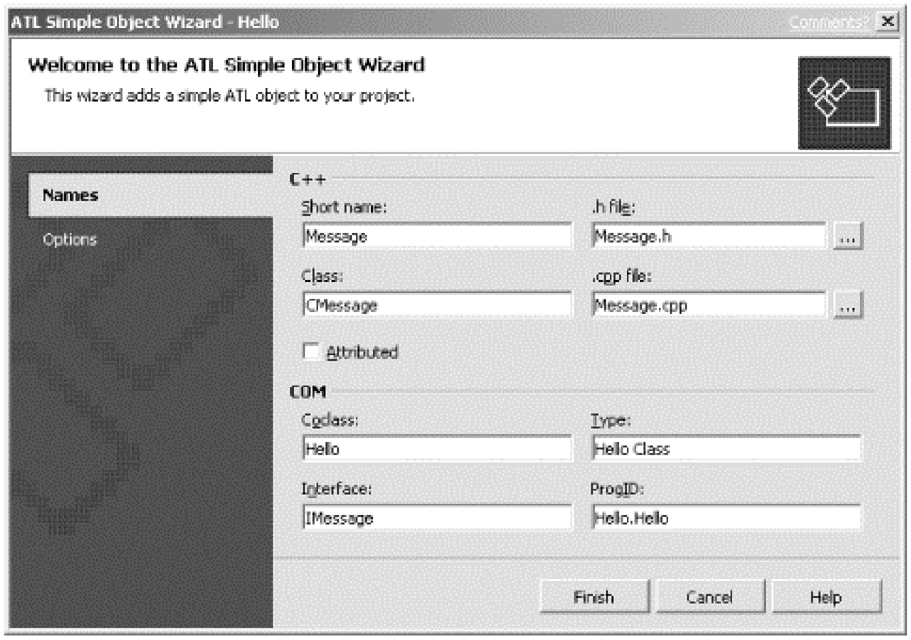
1.3.1 Building a COM Component

We will use ATL 7.0 to generate a classic COM component, although you can also do it in Visual Basic 6.0 with almost the same ease. Start a new ATL project in Visual Studio.NET and name it Hello. For simplicity, do not use Attributed project (deselect Attributed in the ATL Project Wizard under Application Settings). Also, do not select COM+ 1.0 support. This selection adds a few interfaces explained in subsequent chapters that are not relevant to this example. Bring up the Add Class dialog ATL and select the Simple ATL Object item. This step should bring up the ATL Simple Object Wizard dialog (see Figure 1-2). Type the following entries, in order:

1. In the Short Name field, enter **`Message`** .
2. In the CoClass field, enter **`Hello`** .

Your completed dialog should look like Figure 1-2. There is no need to access the Options selection in the dialog (just use the defaults). Click OK when you're done.

Figure 1-2. Use the ATL object wizard to generate a simple COM object


ATL Simple Object Wizard - Hello
Welcome to the ATL Simple Object Wizard
This wizard adds a simple ATL object to your project.
Names
Options
Finish
Cancel
Help

C++
Short name: Message
.h file: Message.h
Class: CMessage
.cpp file: Message.cpp
Attributed [ ]
COM
Coclass: Hello
Type: Hello Class
Interface: IMessage
ProgID: Hello.Hello

Right-click the IMessage interface icon in the Class View, and select Add and then Add Method... from the pop-up context menu. This step brings up the Add Method Wizard. Enter **ShowMessage** as the method name and click OK.

After following these steps, the ATL Object Wizard will generate a new interface definition in the project IDL file, and the new method wizard will add a method to that interface. Verify that the interface definition in the IDL file looks like this:

```
[
   //various IDL attributes
]
interface IMessage : IDispatch
{
    [id(1), helpstring("method ShowMessage")] HRESULT
ShowMessage(  );
};
```

Also make sure that the IDL file contains a type library section with the CoClass definition:

```
[
    //you will have a different CLSID here:
    uuid(C530E78E-9EE4-47D3-86CC-3B4EE39CBD26),
    helpstring("Message Class")
]
coclass Hello
{
   [default] interface IMessage;
};
```

Next, go to the *message.cpp* file and implement the `ShowMessage`( ) method of the `CMessage` class:

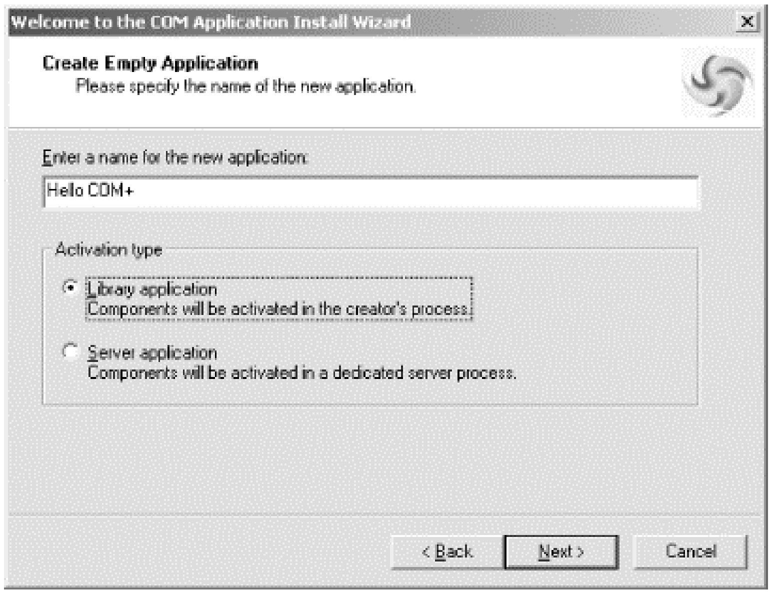
```
STDMETHODIMP CMessage::ShowMessage(  )
{
   ::MessageBox(::GetActiveWindow( ),"Hello COM+","First
COM+ Application",MB_OK);
   return S_OK;
}
```
You can now compile and build the DLL. Every COM+ component must reside in a DLL, and that DLL must contain a type library embedded in it as a resource. ATL will compile and build the DLL for you and add a reference to the type library in the project resource file, the *hello.rc* file. COM+ does not require you to register your component, although the ATL build process will register it for you. As you will see later, COM+ maintains its own components registration and configuration repository.

1.3.2 Creating a COM+ Application

Open the Component Services Explorer and expand My Computer →COM+ Applications folder. Right-click the *COM+ Applications* folder and select New →Application from the pop-up context menu. This step brings up the Application Install Wizard. Click Next on the first wizard screen.

In the next wizard screen, select the Create an Empty Application option in the next wizard screen. Now the wizard will let you specify the new application name and its application type, which can be either a library or a server type (see Figure 1-3). Enter **`Hello COM+`** for the application name, and change the application type from the default Server application to Library application. A *library application* indicates that the components in the application will be loaded directly in the process of their clients (like a classic COM in-proc server). A *server application* indicates that the components will run in their own process (similar to a classic COM local server). You can always change the application name and its activation type later with the Component Services Explorer. Click Next and Finish in the last wizard screen. You have just created your first COM+ application.

Figure 1-3. Naming your new COM+ application and configuring it to be a library or a server application


Welcome to the COM Application Install Wizard

Create Empty Application
Please specify the name of the new application.

Enter a name for the new application: Hello COM+

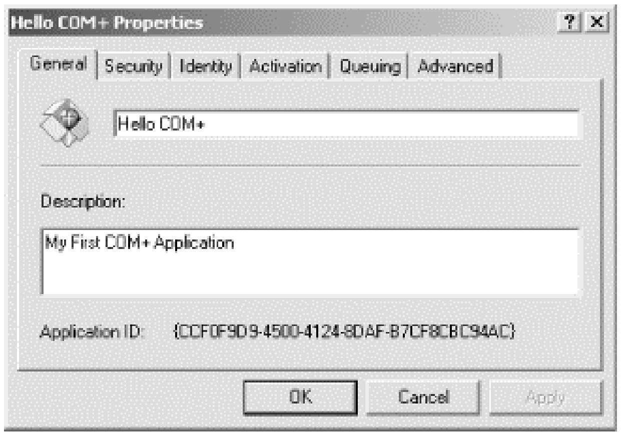
Activation type

[✓ Library application
Components will be activated in the creator's process.]

Server application
Components will be activated in a dedicated server process.

If you examine the *Applications* folder now, you will see your Hello COM+ application. Right-click its icon and select Properties from the pop-up context menu. The application's *properties page*—a collection of tabs that let you configure the application—will now appear. In fact, every item in the Component Services Explorer (applications, components, interfaces, methods, roles, and subscriptions) has a properties page accessible in the same way (by selecting Properties on the item's context menu or the properties button on the Component Services Explorer toolbar). The Hello COM+ application's properties page is shown in Figure 1-4. The General tab contains the application name, which you can change here if you'd like, and a description field. The description field is a useful place to put a few sentences documenting the application's purpose, its owner, etc. Each COM+ application is uniquely identified by a GUID, called the *Application ID*, shown at the bottom of the General tab. You will almost never use the Application ID directly, but COM+ uses it internally.

Figure 1-4. The application properties page


Hello COM+ Properties

Hello COM+

Description: My First COM+ Application

Application ID: {CCF0F9D9-4500-4124-8DAF-B7CF8CBC94AC}

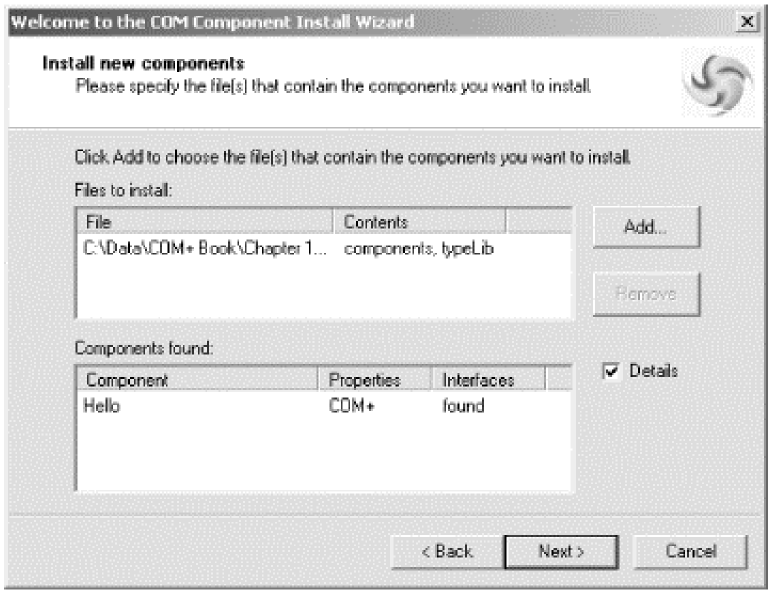
Other tabs on the application properties page let you configure the application activation mode, support for queued components, security settings, and idle-time management. Later chapters describe these application-level configurations in depth.
Close the properties page and examine the application's *Components* folder. As you might expect, it is empty now. You will now add a new component to this application.

1.3.3 Adding a Component to a COM+ Application

You can add a new component to your application (not surprisingly) by using another wizard. Right-click the *Components* folder, select New from the pop-up context menu, and click Component. The Component Install Wizard will now appear. Click Next on the first screen. On the next screen, select Install New Component from the three choices. The wizard will open a standard file-open dialog box. Look for the folder where you built *hello.dll* and select it. The wizard will present you with all the components it could find in the specified DLL. In the case of *hello.dll*, the wizard shows only the single component contained in it (see Figure 1-5). The wizard actually loads the embedded type library in the DLL and looks for CoClass definitions. You can use the Add button to specify additional DLLs. Note that all the components in the selected DLL will be added. If you want to add just a subset of them, you must add them all first and then remove the ones that do not belong in the application manually. Click Next, and then click Finish in the last wizard screen. Your component is now part of the Hello COM+ application.

> Avoid using the "Import component(s) that are already registered" option in the Component Install Wizard. This option has a bug and will not retrieve information about the component(s) interfaces. You will not see the component(s) interfaces and methods in the Component Services Explorer and will not be able to configure them.

Figure 1-5. The Component Install Wizard


Welcome to the COM Component Install Wizard
Install new components
Please specify the file(s) that contain the components you want to install.

Click Add to choose the file(s) that contain the components you want to install.

Files to install:

File	Contents
C:\Data\COM+ Book\Chapter 1...	components, typeLib

Components found:

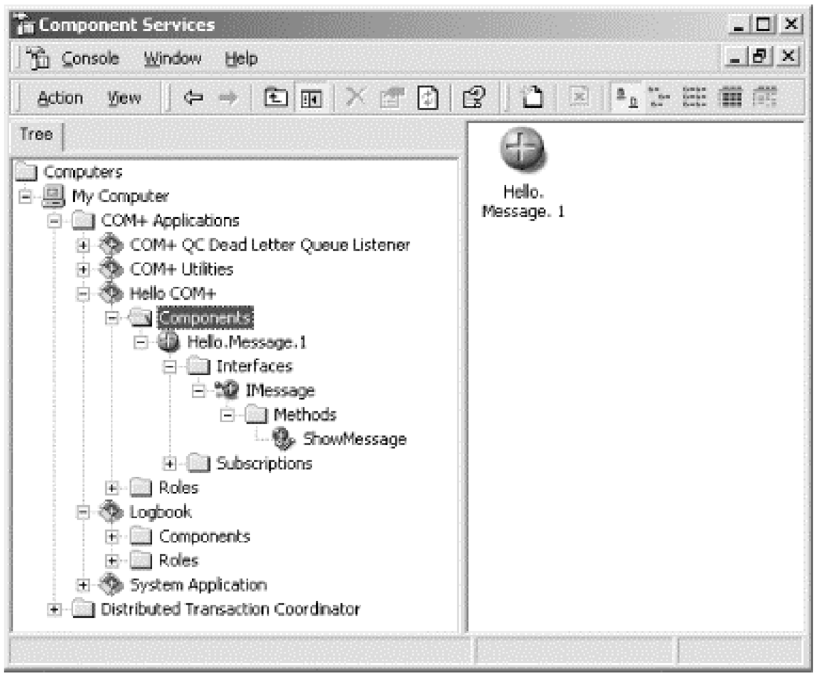
Component	Properties	Interfaces
Hello	COM+	found

[✓ Details]

Add...
Remove
< Back
Next >
Cancel

Because type information is embedded in the DLL, COM+ knows about your component's interfaces and methods. You can expand the *Interfaces* and *Methods* folders (under the Hello.Message component) to verify that COM+ has imported the component correctly. As shown in Figure 1-6, the `IMessage` interface and the `ShowMessage` method were both imported.

Figure 1-6. The Hello COM+ application and its contained component



The *Interfaces* folder contains one entry for each interface your component supports. The interfaces on the CoClass definition in the type library determine the number of entries. The *Methods* folder contains one item for each method in that interface, again based on the interface definition in the type library.

1.3.4 Writing a Test Client

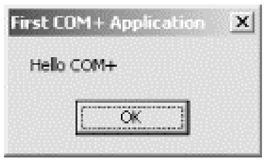
Clients can create the component using the class ID `CLSID_Hello` (C++) or `Hello` (Visual Basic 6.0). Although the component is now a COM+ component and is part of a COM+ application, the client-side code is the same as if the component were still a classic COM component. To prove this point (and test your component), write a short C++ client, such as the code in Example 1-1.

Example 1-1. A simple COM+ client

```
#import "Hello.dll" no_namespace named_guids
::CoInitialize(NULL);

HRESULT hres = S_OK;
IMessage* pMessage = NULL;

hres = ::CoCreateInstance(CLSID_Hello,NULL,CLSCTX_ALL,

IID_IMessage,(void**)&pMessage);

hres = pMessage->ShowMessage(  );
pMessage->Release(  );

::CoUninitialize(  );
```

When you run the client, you will see the "Hello COM+" message box (see Figure 1-7).

Figure 1-7. The "Hello COM+" message box from your first COM+ component


First COM+ Application

Hello COM+

Alternatively, you can write the client side in Visual Basic 6.0. Add the component type library `Hello.TLB`, the Visual Basic project references browser, and write:

```
Dim obj As Hello

Set obj = New Hello
obj.ShowMessage
set obj = Nothing
```

Visual Basic 6.0 clients can also create the object using its prog-ID. In that case, the type-library is not required (at the expense of type-safety):

```
Dim obj As Object
Set obj = CreateObject("Hello.Message.1")
obj.ShowMessage
set obj = Nothing
```

Because the client side remains constant, regardless of the component configuration and application type, COM+ helps decouple the client from the server. This point is discussed in depth in the next chapter.

1.4 COM+ Configured Components

COM+ allows you to import only in-proc (DLL) components. You cannot import COM components that reside in a local server (EXE); COM+ lets you configure the activation type of your application, server, or library. In the case of a library, the client simply loads the original DLL into its process and uses the component. If you configure the application to be a server application, COM+ promotes your original DLL to become a local server by hosting it in a surrogate process of its own. However, COM+ cannot make a library application out of a COM local server. In addition, many COM+ services require explicit process-level administration that the local server's code simply does not contain.

Once an in-proc component is imported to COM+, it is called a *configured component* to emphasize the fact that much component functionality and behavior is actually configured and administered outside the component. A classic COM component (be it in-proc or local) that has not been imported into COM+ is called a *nonconfigured component.* Configured and nonconfigured components can interact freely and call each other's interfaces. The configured component must reside on a Windows 2000 machine, but the client of a configured component can reside on any Windows-family machine, such as Windows NT, Windows Me, or Windows 9x.

Configuration lets you control the way your application, component, interface, or method behaves under COM+. The COM+ development paradigm lets COM+ manage as much of the nonbusiness logic plumbing as possible by declaring what services you want to use. Doing so lets you focus on the domain problem you are trying to solve and add business value instead of plumbing code to your product.

Your configured component's interfaces can be dual, dispatch, or custom interfaces. If you use automation-compliant interfaces, you do not need to provide COM+ with a proxy/stub DLL (see COM Interface Types for more information).

COM Interface Types

In general, there are two kinds of COM interface types: automation-compliant interfaces and custom interfaces. Contrary to common conceptions, an automation-compliant interface does not have to derive from IDispatch or have all the parameters be variants or variants-compatible types (such as a BSTR or long). An automation-compliant interface must have one of the following two directives in its definition: dual or oleautomation. For example:

```
[
   object,
       uuid(30548235-4EC3-4087-9956-ED26748F47E9),
       dual,
       helpstring("An example for automation
compliant interface"),
]
interface IMyInterface : IUnknown
{
   HRESULT MyMethod([in]long lNumber);
};
```

COM can marshal an automation-compliant interface by creating the appropriate proxy and stub automatically at runtime. However, automation-compliant interfaces do have limitations on parameter types; for example, they cannot have as method parameters structs with pointers in them. For ultimate flexibility, you can use custom interfaces. These interfaces do not have dual or oleautomation in their interface definition, and it is the developer's responsibility to provide a proxy and a stub DLL.

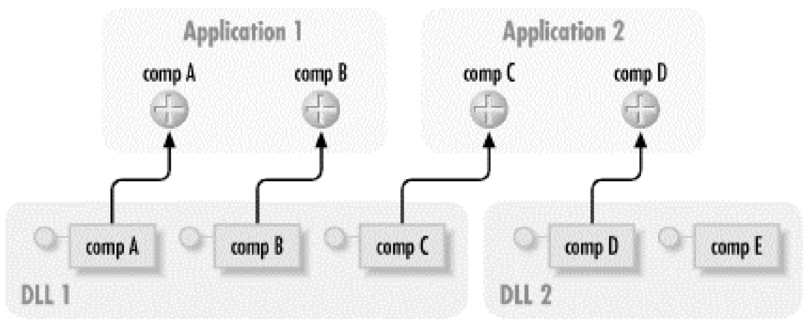
However, if your design calls for custom interfaces, you should provide COM+ with a proxy/stub DLL that was built using the MIDL switch `/Oicf` to enable type library marshaling. In any case, configured components cannot use interfaces that require custom marshaling. You can develop configured components in C++, Visual Basic, or even C#, since one of the core principles of COM, language independence, is maintained in COM+.

You may be wondering by now, where does COM+ store the configuration information for all your applications and components? Unlike classic COM, COM+ does not use the Windows registry. COM+ uses a dedicated repository called the *COM+ catalog.* No formal Microsoft documentation of the exact physical location of the catalog exists, simply because it is not useful to you. The only bit of configuration information still stored in the Windows registry is the component threading model and remaining classic COM information (such as `InprocServer32` and `prog-ID` registry keys).

1.5 Applications, DLLs, and Components

COM+ applications are logical packaging units; DLLs, however, are physical packaging units. There is no correlation between logical and physical packaging. The only requirement is that a configured component must belong to exactly one COM+ application; it cannot belong to more than one, and it must belong to at least one to take advantage of COM+ component services. As demonstrated in Figure 1-8, a COM+ application can host components from one or multiple DLLs (Application 2 has components from two DLLs). It is also possible that not all the components in a DLL are hosted in COM+ applications (such as component E), and one DLL can contribute components to multiple COM+ applications (DLL 1 contributes components to Application 1 and Application 2).

Figure 1-8. COM+ applications and DLLs



The separation of physical from logical packaging gives you great flexibility in designing your application's layout. All the components in the same COM+ application share the same application-level configuration settings, regardless of their underlying DLL packaging. However, I recommend that you avoid installing components from the same DLL into more than one application, such as components B and C in Figure 1-8. The reason is that components in the same application are assumed to operate tightly together and trust each other. On the other hand, nothing is assumed about components from different applications. By placing components from the same DLL into multiple applications, you may introduce needless security checks. You might also introduce cross-process marshaling overhead, if those components need one another to operate, which is probably why they were put in the same DLL in the first place. The COM+ Component Install Wizard also does not handle components from the same DLL in multiple applications very well. When you use the wizard to add components from a DLL to an application, the wizard tries to add all components in the DLL to the application. If some of the components are already part of other applications, the wizard will treat this situation as an error since it

will think you are trying to include a component in more than one application.
The bottom line is that you should put all components that cooperate closely or perform related functionality into a single application. Those components can be written by multiple developers and be contained in multiple DLLs, but they will ultimately share the same application configuration and be deployed together.

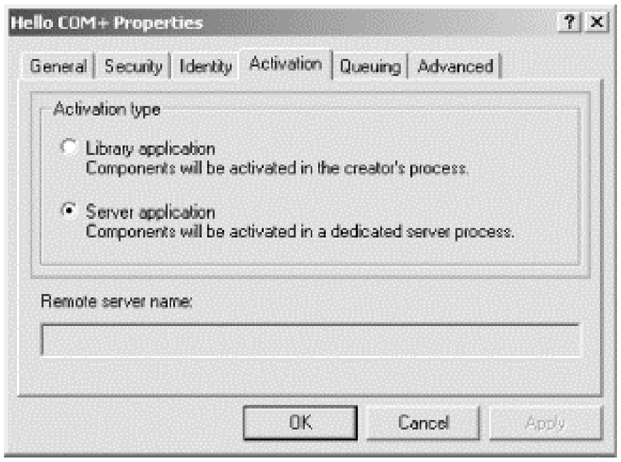
1.6 Configuring COM+ Applications

The primary benefit of using COM+ is that you can configure a component or the application containing it without changing any code on the object or the client side. This advantage enables you to focus your object code on its intended purpose, relying on the various services COM+ provides instead of having to develop them yourself. This section shows you how to configure some of the application-level options for the Hello COM+ program you created.

1.6.1 COM+ Application Types

As mentioned previously, the application activation type (a server or a library application) is a configurable application-level attribute called *activation*. You can configure the application's activation type in the application's properties page, under the Activation tab (see Figure 1-9).

Figure 1-9. Application Activation tab



Activation type

[ ] Library application
Components will be activated in the creator's process.

[✓ Server application]
Components will be activated in a dedicated server process.

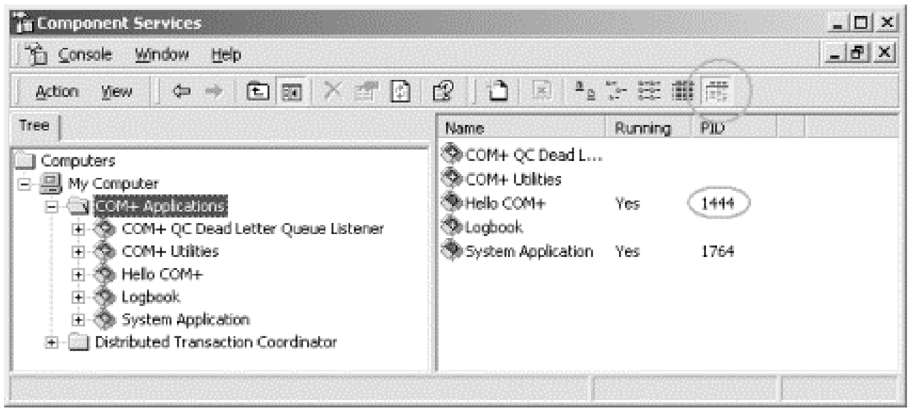
Remote server name:

Changing the application type has significant implications for most COM+ services. The application type is a design-time decision that should consider the security needs of your components, the calling patterns of your clients, fault isolation (a server application gets its own process), and specific COM+ services requirements. Throughout the book, a particular service configuration that is related to the activation type is pointed out explicitly. However, even without knowing much about COM+, you can use the following rule to decide on your activation type: prefer server type applications, unless you absolutely need to run in the client process for performance reasons. Library applications have some limitations in using COM+ services (such as security and queued component support), and they cannot be accessed from another machine.

1.6.2 COM+ Surrogate Processes

If the original COM components resided in a DLL, how does COM+ achieve different activation modes for the configured components? When the application is configured as a library, the client loads the DLL directly into its process. When the application is configured as a server application, COM+ creates a surrogate process for it, called *dllhost.exe,* that loads the DLL. COM+ then places a proxy in the client process and a stub in the surrogate process to connect the client to the object. You can have multiple instances of the dllhost process running on your machine simultaneously; if clients have created objects from different server applications, each server application gets its own instance of dllhost.
To verify these points yourself, configure the Hello COM+ example to run as a server application. Run the test client again, create the object, and call the `ShowMessage( )` method, but do not press the OK button. The Component Services Explorer gives you visual feedback when a server application is running: the application icon and the active components will be spinning. Library applications will have no visual feedback when they are running in a client process, even if that process is another COM+ server application. Expand the *COM+ Applications* folder and select the Status View on the Component Services Explorer toolbar (the button at the far right end of the toolbar; see Figure 1-10). The Component Services Explorer will display the process ID of the running server applications. Record the process ID for the Hello COM+ application. Next, bring up Windows Task Manager and locate the process with a matching ID. Its image name will be *dllhost.exe*.

Figure 1-10. Examining a running server application



Computers
My Computer
COM+ Applications
COM+ QC Dead Letter Queue Listener
COM+ Utilities
Hello COM+
Logbook
System Application
Distributed Transaction Coordinator

Name	Running	PID
COM+ QC Dead L...		
COM+ Utilities		
Hello COM+	Yes	1444
Logbook		
System Application	Yes	1764

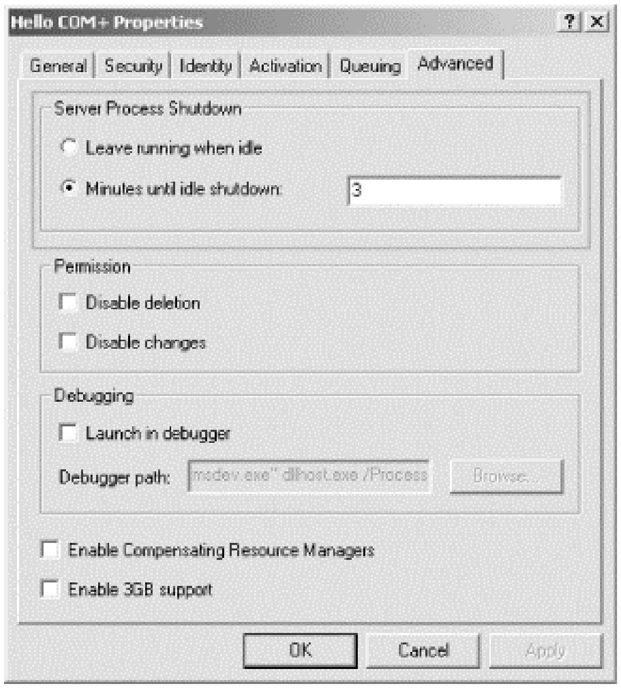
The first `CoCreateInstance( )` request for a component in a server application creates a new dllhost process, to host components from that application only. Subsequent `CoCreateInstance( )` calls to objects from the same application create new objects in the existing dllhost instance. Unlike classic COM, there is no way to create each object in its own process. No COM+ equivalent to the COM call you make to `CoRegisterClassObject(...REGCLS_SINGLEUSE...)` exists.

The Component Services Explorer also lets you manage server application activation administratively. You can shut down a running application by right-clicking on its icon in the Component Services Explorer and selecting Shutdown from the pop-up context menu. You can shut it down even when clients are holding active references to objects in the application. (You shut down applications this way frequently during debugging sessions.) The Component Services Explorer does not provide a way to shut down library applications, since COM+ may not even manage their client process. You can also select Start from the server application pop-up context menu to launch a new dllhost process associated with that application. However, no objects will be created unless you use object pooling, which is discussed in Chapter 3.

1.6.3 Idle Time Management

Another distinction between a classic COM local server and a COM+ server application is process shutdown. In classic COM, when the last client has released its last reference on an object in the process, COM would shut down that process. COM+ provides *idle time management* for COM+ server applications. COM+ server applications can be left running indefinitely even when idle (when there are no external clients), or you can have COM+ shut them down after a predetermined timeout. This shutdown is done for the sake of performance. Imagine a situation in which a client creates an object from a server application every 2 minutes on average, uses it for 1 minute and 55 seconds, and then releases it. Under classic COM, you would pay an unnecessary performance penalty for creating and destroying the server process. Under COM+, you can configure the server application to be left running when idle for a specific time. If during that time no client request for creating a new object has come through, COM+ is allowed to shut down the process to release its resources. In this example, you would perhaps configure the server application to be left running when idle for 3 minutes, as you would want to compensate for variances in the client calling pattern. If a new call comes in within those 3 minutes, COM+ zeros the idle time counter and starts it up when the application is idle again. You can configure server application idle time management under the Advanced tab on the server's properties page (see Figure 1-11). Library applications do not have an idle time management option and will be unloaded from their client process once the last object has been released.

Figure 1-11. Configuring server application idle time management



Server Process Shutdown
- Leave running when idle
- Minutes until idle shutdown: 3 (selected)

Permission
- [ ] Disable deletion
- [ ] Disable changes

Debugging
- [ ] Launch in debugger
- Debugger path: msdev.exe" dllhost.exe /Process

- [ ] Enable Compensating Resource Managers
- [ ] Enable 3GB support

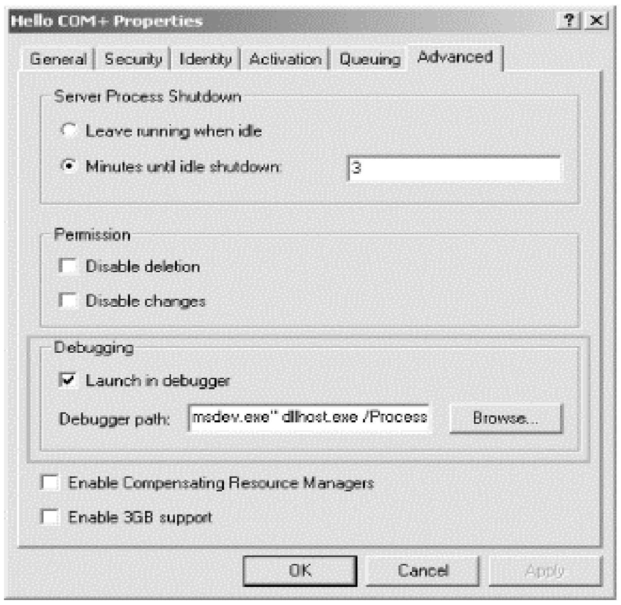
1.7 Debugging COM+ Applications

Debugging a COM+ application, be it a library or a server application, is not much different from debugging an in-proc COM object or a local server. A library application has the clear advantage of allowing you to step through your code directly from the test client, since a library and a server application share the same process. A server application always runs in a different process than your test client and, therefore, in a different debug session (a different instance of Visual Studio is attached to that process). When debugging the business logic part of your application, you may find it useful to debug it as a library application, even if the design calls for a server application. When debugging a library application, you may also need to point Visual Studio to the exact location of the component's DLLs. This step is required so you can set breakpoints in the component's code. When debugging a component in a server application, you can step into the component's code from the test client side in two ways. First, you can start the client project in the debugger, break at a line where you call a method on a component in the server application, and simply step into it (F11 in Visual C++ or F8 in Visual Basic). This process launches a new instance of the debugger and attaches it to the running dllhost containing your component. You can then step through your component's code. Second, you can attach a debugger to a server application by configuring it to launch in a debugger. On the server application properties page, under the Advanced tab, there is the Debugging properties group. If you check the Launch in debugger checkbox (see Figure 1-12), when the first request for creating an object from that application comes in, COM+ launches the application in a Visual C++ debugger session. You may use this option often to track bugs in the constructors of components or bugs that do not happen in the scope of a client call. COM+ is able to attach the debugger to the application using a command-line option for Visual Studio. When you launch the debugger with an executable filename as a parameter, the debugger starts a debug session and creates the specified process (in COM+'s case, always dllhost). COM+ also specifies the server application ID as a command line parameter for dllhost:

```
msdev.exe dllhost.exe /ProcessID:{CCF0F9D9-4500-4124-
8DAF-B7CF8CBC94AC}
```

This code informs dllhost that it is now associated with the specified server application.

Figure 1-12. Launching COM+ server application in a debugger



Server Process Shutdown

- ( ) Leave running when idle
- (•) Minutes until idle shutdown: 3

Permission

- [ ] Disable deletion
- [ ] Disable changes

Debugging

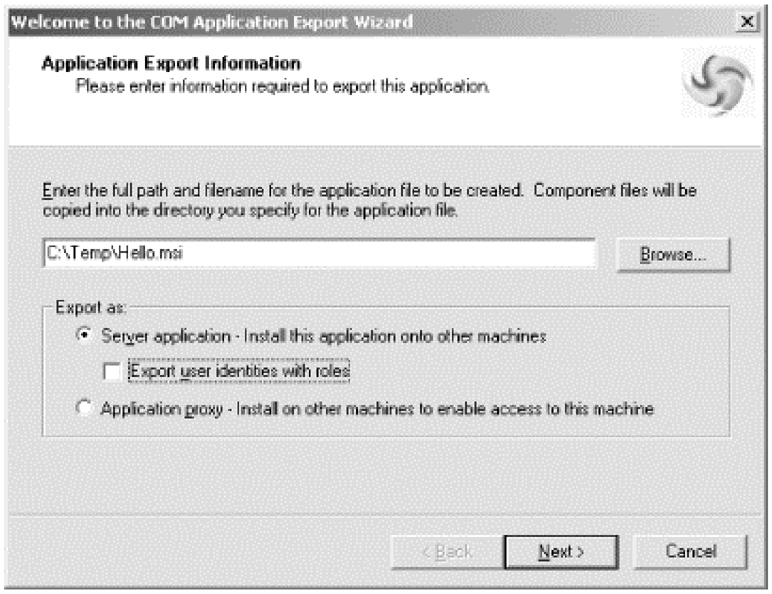
- [✓] Launch in debugger
- Debugger path: msdev.exe" dllhost.exe /Process

- [ ] Enable Compensating Resource Managers
- [ ] Enable 3GB support

1.8 Deploying COM+ Applications

Once you have tested your COM+ application and configured all the COM+ services to your liking, you need to install your application on a customer/client machine. The Component Services Explorer can generate a special file that captures all your application components and settings. This file is a Windows Installer (MSI) file, identified by the *.msi* file extension. Clicking on an MSI file launches the Windows Installer and installs the application with all its COM+ configuration parameters. There is a one-to-one relationship between an application and an MSI file. Thus, if you have multiple applications in your product, you must generate one MSI file for each application.
To generate the MSI file, right-click on your application icon in the Component Services Explorer and select Export from the pop-up context menu. This action should bring up the Application Export Wizard. Click Next to go to the second wizard screen, where you are requested to enter the name and location for the application export file to be created (see Figure 1-13). Next, you should decide how to export the application: as a Server application or as an Application proxy (see Figure 1-13). Click Next and then click Finish on the next Wizard screen.

Figure 1-13. Exporting a COM+ application


Welcome to the COM Application Export Wizard

Application Export Information

Please enter information required to export this application.

Enter the full path and filename for the application file to be created. Component files will be copied into the directory you specify for the application file.

C:\Temp\Hello.msi

Export as:

(●) Server application - Install this application onto other machines

[ ] Export user identities with roles

( ) Application proxy - Install on other machines to enable access to this machine

1.8.1 Proxy COM+ Applications

The names *Server application* and *Application proxy* are confusing. A "Server application" export is relevant for both library and server applications. It means that the application will include in the MSI file the COM objects themselves, their settings, and their proxy/stub DLLs (if required), and will install all on the server machine.
An "Application proxy" export installs on the client machine only the type information in the MSI it creates (as well as the proxy/stub DLLs, if required). The generated file does not have to include the components themselves (unless the type information is embedded in the components, in which case the components are only used as containers and are not registered). You can use a proxy installation when you want to enable remote access from a client machine to the machine where the application actually resides. A proxy export is available only for a COM+ server application, not for a library application.
When you install a server export on another machine, it will install the components for local activation. `CoCreateInstance( )` requests create the objects locally—in the client process, if it is a library application, or in a dllhost process, if it is a server application.
When you install a proxy export, activation requests on that machine will be redirected to another remote machine. In a way, a proxy export installed on a client machine is a third kind of COM+ application. This kind is usually called a *proxy application*. You can configure the proxy application to access any remote machine on the network where the server application is installed, not just the machine that generated the proxy export. You specify the "real" application location on the proxy application properties page under the Activation tab.

A proxy application can even be installed on machines running Windows NT or Windows 9x with DCOM, provided those machines have Windows Installer installed on them. Because the Windows Installer cannot use the COM+ catalog to store the proxy application information on a non-Windows 2000 machine, it will use the registry and will store only the subset of information required for DCOM there. Windows Installer is not commonly found on non-Windows 2000 machines. To make sure clients on those machines are able to access your COM+ applications, you should incorporate the Windows Installer installation in your product installation. The Windows Installer installation file is called *instmsi.exe* and is available as part of the Developers Platform SDK.
A proxy application cannot export another MSI file. In fact, all the application-component, interface, and method-level settings on a proxy application are disabled, except the Remote server name under the Activation tab. The Remote server name edit box is disabled in library and server applications.

1.8.2 Installing and Uninstalling an Exported Application

The most common way to install an MSI file on another machine is simply to click on it, which will launch the Windows Installer. The application files (DLLs and proxy/stubs) will be placed in a default location:

```
\Program Files\COMPlus Applications\{<the application's guid>}
```

If you wish to have the application installed in a different location, you must use the Component Services Explorer Application Install Wizard. Bring up the wizard and select Install pre-built application(s). Browse to where the MSI file is stored, and select it. The wizard will let you choose whether you want to use the default location for installation or specify a different one.
If you want to automate uninstalling COM+ applications, you can use a command line instruction to invoke the Windows Installer to uninstall a COM+ application:

```
msiexec -x <application name>.msi
```

You can also use the Windows Control Panel's Add/Remove Programs applet to add or remove COM+ applications.

1.9 Summary

In this chapter, you created a trivial example COM component and implemented it in a DLL. You used it as an in-proc server or as a local server and even controlled its life cycle and idle time management by configuring the component (actually its containing application) differently. All this was achieved without changing a single line of code on the object or the client side. This achievement reflects the power of COM+: it enables you to focus on your product and domain problems at hand, while declaratively taking advantage of available services. The rest of this book discusses these services thoroughly, including their interactions and pitfalls, and provides tips and tricks for how to apply them productively.

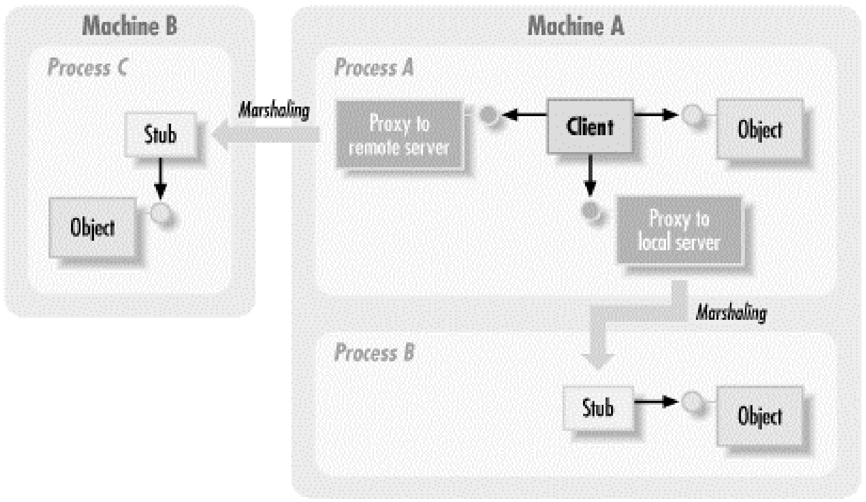
Chapter 2. COM+ Context

COM+ provides services to components by intercepting the calls the client makes to component interfaces. The idea of providing services through an interception mechanism is not a COM+ innovation. As you will see, classic COM also provides component services via interception. What is new is the length to which COM+ takes the idea. This chapter starts by describing the way classic COM uses marshaling to provide its services and to encapsulate the runtime requirements of its objects. Next, the chapter introduces you to the *COM+ context*—the innermost execution scope of an object. COM+ call interception occurs at context boundaries. Generally, you need not be concerned with contexts at all. They are transparent to you, whether you develop a client or a component. However, the COM+ context is a good model for explaining the way COM+ services are implemented. This book clearly outlines the few cases when you should interact with the contexts directly. Interaction with the contexts occurs mostly when dealing with COM+ instance management and transactions, but also when dealing with some security issues.

2.1 Encapsulation via Marshaling in COM

One of the core principles of classic COM is *location transparency.* Location transparency allows the client code to be independent of the actual object's location. Nothing in the client's code pertains to where the object executes, although the client can insist on a specific location as well. A client CoCreates its objects and COM instantiates them in the client's process, in another process on the client's machine, or on another machine altogether. COM decides where the objects will execute based on a few Registry values. Those values are maintained outside the object code. A change in those values can cause the same object to be activated in a different location. The same client code handles all cases of object location. You can say that COM completely encapsulates the object location. A key idea in object-oriented and component-oriented programming is *encapsulation,* or information hiding. Encapsulation promotes the design of more maintainable and extensible systems. By ignoring the object location, the client code is decoupled further from the object. The client code does not need to be modified if the object location changes. COM encapsulates the object location by introducing a proxy and stub between the object and its client. The client then interacts with the object directly or through a proxy, and COM marshals the call from the client to the object's true location, if it needs to (all three cases are shown in Figure 2-1). The important observation here is that the client code is not required to make assumptions about the location of its called objects or to make explicit calls across processes (using named pipes, for instance) or across machines (using sockets).

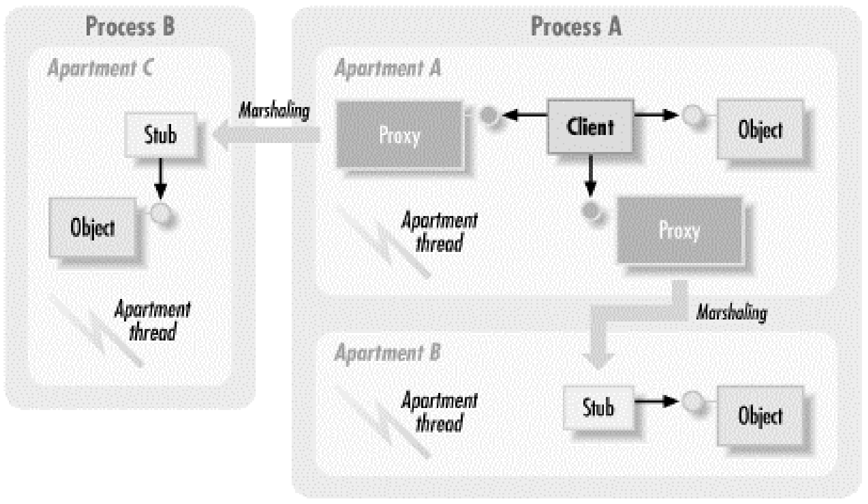
Figure 2-1. Classic COM completely encapsulates the object location from the client by introducing a proxy/ stub between them



To provide location transparency, COM proxies are polymorphic with the object; they support exactly the same set of interfaces as the real object, so the client cannot tell the difference between the proxy and the real object.
Another time when classic COM encapsulates an object property using marshaling is in its handling of the object's synchronization needs. The object's developer declares in the Registry what threading model the object uses. If an incompatibility exists between the creating client-threading model and the object's threading model, COM puts a proxy and stub between them and marshals calls from the client thread to the object thread. Since many threads can exist in a given process, COM divides a process into *apartments,* and any call crossing an apartment boundary is marshaled (see Figure 2-2). Again, the proxy and stub completely encapsulate the object's execution thread. The same client code can handle calling methods on objects on the same thread (in the same apartment), on a different thread (in a different apartment) in the same process, or on another thread in a different process. The proxy and stub are responsible for performing a thread context switch when marshaling the call from the client thread to the object thread. Because the object needs to appear to the client as though it is executing on the same thread as the client, the proxy and stub will also handle the required synchronization; the proxy has to block the client thread and wait for the stub to return from the call on the object thread. COM concurrency management makes it possible for

the client to ignore the exact synchronization requirement of the object. A dedicated synchronization protocol, such as posting messages between the client and the object, or signaling and waiting on events or named events is not necessary. Because nothing in the client's code considers the object's threading need, when the object's threading model changes (when a new version of the object with a new threading model is deployed), the client code remains unchanged.

Figure 2-2. Classic COM encapsulates the object execution thread by inserting a proxy and a stub between the client and the object



The two examples have a few things in common. The proxy intercepts calls from the client to the object, making sure the object gets the runtime environment it requires to operate properly. The proxy and stub marshal away incompatibilities between the client and the object, and they perform pre- and post-call processing, such as thread context switching, cross-process communication, blocking the calling thread, and signaling internal events. In both examples, the object declares its requirements in the Registry, rather than providing specific code for implementing them.

While classic COM provides only a few services by intercepting the client's calls, you can see the potential for implementing additional services through this mechanism. Ideally, you could declare which services your component requires and then use system component services instead of implementing them yourself. This is where COM+ comes in.

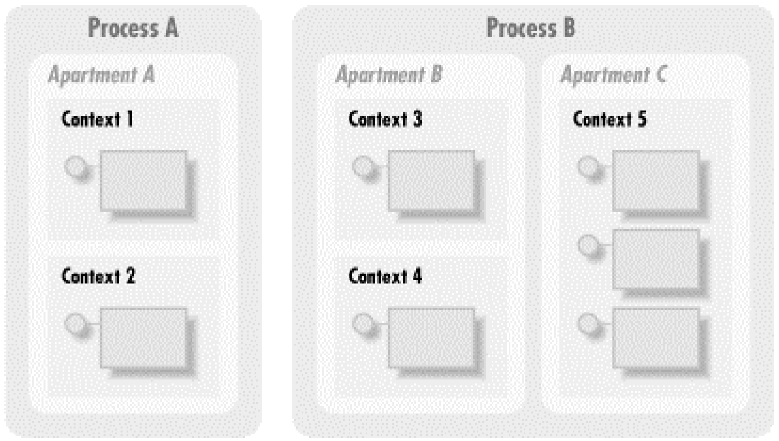
2.2 Encapsulation via Interception in COM+

COM+ provides its component services via interception. You can configure your component to take advantage of services, and COM+

puts a proxy and stub between the component and its client, if the client and the component instance are incompatible with any one of the services. It also puts a proxy and stub between them if a service requires interception, regardless of the way the client and the object are configured. The exact object configuration is completely encapsulated by the proxy and stub and the call interception. Nothing in the client code couples it to the object configuration. This development is a major step toward ultimate encapsulation, in which the component contains almost nothing but business logic and in which the way it uses component services such as transactions, security, events, and activation is hidden from the client. Similarly, the component does not care about its client configuration, as the two do not need to interact with each other about the way they use the services.

Because an object can have the same threading model as its creating client while differing in other service configuration, apartments can no longer be the innermost execution scope of an object. Instead, COM+ subdivides apartments, so each object can be placed in a correct runtime environment appropriate to its needs and intercept all calls to the object. The subdivision of an apartment into units of objects that share the same configuration is called a *context.* Each apartment has one or more contexts, and a given context belongs to exactly one apartment. A context can host multiple objects, and each object belongs to exactly one context. Figure 2-3 shows an example of how processes and apartments can be broken down into contexts under COM+.

Figure 2-3. COM+ subdivides apartments into contexts



Because a COM+ object must belong to exactly one context, every apartment has at least one context and potentially many more. There is no limitation to the number of contexts an apartment can host. All calls in and out of a context must be marshaled via a proxy and stub so that COM+ can intercept the calls and provide configured services. This idea is similar to the classic COM requirement that all cross-apartment calls be marshaled so that

COM can enforce threading model configurations. Objects in the same context can have direct pointers to one another, because they are configured to use the same set of services in a way that allows same-context activation, and hence, direct access. Mediating between objects in the same context is not necessary.

2.2.1 Lightweight Proxies

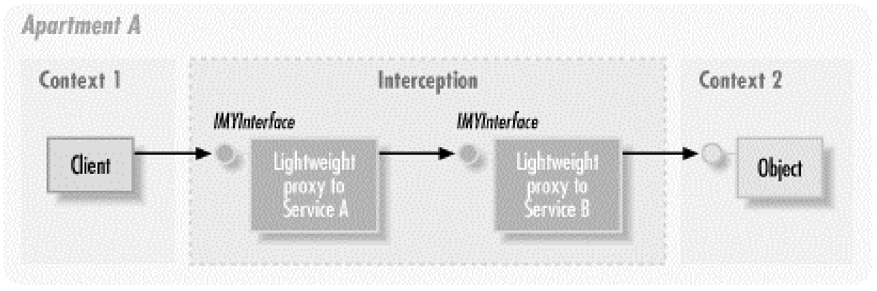
When COM+ marshals a call between two contexts in the same apartment, it does not need to perform a thread context switch. However, COM+ still puts a proxy and stub in place to intercept the call from the client to the object and perform a *service context switch.* This switch ensures that the object gets the runtime environment it requires. COM+ uses a new kind of proxy for this marshaling: a *lightweight proxy*. It is called a lightweight proxy because no expensive thread context switch is needed to marshal calls from the client to the object. The performance hit for a service context switch is a fraction of that incurred when performing a thread context switch. A service context switch can sometimes be as lightweight as simply checking the value of a flag, but usually it involves some pre- and post-call processing to marshal away differences in the runtime environment between the client and the object.

The lightweight proxies are not the standard proxies used for cross-apartment/process/machine calls. Standard proxies are either created using the MIDL compiler or provided by the standard type library marshaler. For a service switch, COM+ generates the lightweight proxies on the fly, at runtime, based on the exact object configuration. A lightweight proxy, like any other proxy, presents the client with the exact same set of interfaces as those found on the actual object. COM+ provides the lightweight proxy with the right interface signatures based on the type library embedded in the component's DLL.

An example for a lightweight proxy is a proxy that provides calls synchronization to the object. If the object is configured to require synchronization (to prevent access by multiple concurrent threads), but its client does not require synchronization, COM+ puts a lightweight synchronization proxy between the two. Another example is security. If the object is configured to require an access check before accessing it, verifying that the caller was granted access to the object, but its client does not care about security, there will be a lightweight security proxy in between. This proxy makes sure that only authorized callers are allowed access to the object

If the object is in a different context from that of its caller because of incompatibility in just one component service (or if a service always mandates a separate context), there will be just one lightweight proxy between the caller and the object. Therefore, what should COM+ do if the client and the object differ in more than one service? The exact way the lightweight proxies mechanism is implemented is not documented or widely known. However, in this case, COM+ probably does not generate just one lightweight proxy to do multiple service switches, but rather puts in place as many lightweight proxies as needed, one for every service switch. For example, consider an object that implements the interface `IMyInterface` and is configured to use two COM+ services: Service A and Service B. If the client does not use Service A and Service B, COM+ puts two lightweight proxies in place, as shown in Figure 2-4. The lightweight proxy to Service A only knows how to do a Service A switch, and the lightweight proxy to Service B only knows how to do a Service B switch. Both services support the `IMyInterface` interface, and would delegate the client call from the first proxy to the second, to the object, and then back again. The net result is that when the client calls into the context where the object resides, the object gets the correct runtime environment it requires to operate. If the client and the object both use Service C, no lightweight proxy to Service C is required. (Stubs have been removed from Figure 2-4 for clarity.)

Figure 2-4. Lightweight proxies perform service switches



2.2.2 Assigning Objects to Contexts

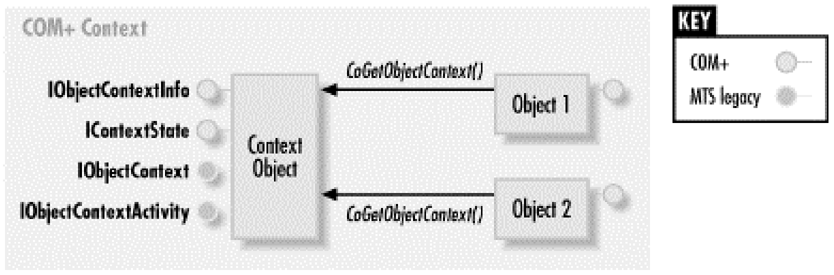
When a client calls `CoCreateInstance( )` (`New` or `CreateObject( )`, in Visual Basic), asking for a new instance of a configured component (an object), COM+ first constructs the object and then decides which context to place the object in. In COM+ terminology, COM+ decides in which context to activate the object. COM+ bases its decision on two factors: the component's configuration and the configuration of its creating client. Obviously, it would be best if the object could share a context with the client. Doing so would obliterate the need for COM+ to marshal calls from the client to the object, and thus avoid having to pay even the slight performance penalty of lightweight proxies.
COM+ examines the newly created object's configuration in the COM+ catalog and compares it with the configuration (or rather, the

context attributes) of the creating client. If the client's context can provide the object with a sufficient runtime environment for its configuration, COM+ places the object in the client's context.
If, on the other hand, the client's context cannot provide the object with its required runtime environment, COM+ creates a new context, places the object in it, and puts lightweight proxies between the two contexts. Note that COM+ does not try to find out if another appropriate context for the object in that apartment already exists. The algorithm is simple—the object either shares its creator's context or gets a new context. Obviously, the precondition for same-context activation is having a compatible threading model between the client and the object. Otherwise, the object is placed in a different apartment, and hence, a different context by definition, since a context belongs to exactly one apartment.
Classic COM components (nonconfigured components) do not rely on COM+ services to operate and do not require lightweight proxies to mediate between their client runtime environment and their own. If a nonconfigured component can share the same apartment as its creating client (compatible threading model), it will also share its context, and the client will get a direct pointer to it, instead of a proxy. However, if the nonconfigured object requires a different apartment, it is placed in a suitable apartment, in what is known as the *default context.* Each apartment has one default context used for hosting nonconfigured components. The default context is defined mostly for COM+ internal consistency (every object must have a context), and no lightweight proxies are used when objects in other contexts (in the same apartment) access it.
You can sum up the COM+ algorithm for allocating objects to contexts with this rule: a configured component is usually placed in its own context, and a nonconfigured component shares its creator's context.

2.3 The Context Object

COM+ represents each context by an object called the *context object.* Every context has exactly one context object. Objects can obtain a pointer to their context object by calling `CoGetObjectContext( )` (see Figure 2-5). All objects in the same context get the same context object.
`CoGetObjectContext( )` is defined as:

Figure 2-5. By calling CoGetObjectContext( ), objects can get a pointer to their context's context object



```
HRESULT CoGetObjectContext(REFIID riid, void**
ppInterface);
```

The context object supports a few interfaces, so the first parameter of `CoGetObjectContext( )` is always an IID that specifies which interface to retrieve. Two of the context object's interfaces, `IObjectContext` and `IObjectContextActivity`, are legacy interfaces from MTS and are provided primarily for backward compatibility with MTS components running under COM+. The other two interfaces, `IContextState` and `IObjectContextInfo`, are specific to COM+. Throughout this book, all chapters use these two interfaces, rather than the legacy MTS interfaces.

Programming in the COM+ Environment

To make programmatic calls in C++ against COM+-specific interfaces, such as `IObjectContextInfo`, you need to install the latest Platform SDK and include the header file *comsvcs.h* (from the SDK include directory, not the Visual Studio 6.0 include directory) or import the DLL *comsvcs.dll* from your system directory and provide the following import directives:

```
#import "COMSVCS.DLL"
raw_interfaces_only,raw_native_types,
                      no_namespace,named_guids,
                      no_auto_exclude
```

Visual Basic 6.0 developers should import the COM+ Services Type Library to access COM+ services programmatically.

The `IContextState` interface controls object deactivation (discussed in Chapter 3) and transaction voting (discussed in Chapter 4) by manipulating state bits in the context object.

`IObjectContextInfo` gains access to various aspects of the current transaction, retrieves the current activity ID (discussed in Chapter 5), and retrieves the current context ID. The `IObjectContextInfo` interface is defined as:

```
interface IObjectContextInfo : IUnknown
{
   BOOL IsInTransaction( );
   HRESULT GetTransaction([out]IUnknown** ppTransaction);
   HRESULT GetTransactionId([out]GUID* pTransactionId);
   HRESULT GetActivityId([out]GUID* pActivityId);
   HRESULT GetContextId([out]GUID* pContextId);
};
```

Every COM+ context has a unique ID (a GUID) associated with it. Retrieving the current context ID is sometimes useful for tracing and debugging purposes. Example 2-1 shows how to trace the current context ID by calling `CoGetObjectContext( )`, requesting the `IObjectContextInfo` interface, and then calling the `IObjectContextInfo::GetContextId( )` method.

Example 2-1. Retrieving the current context ID with IObjectContextInfo::GetContextId( )

```
HRESULT hres = S_OK;
IObjectContextInfo* pObjectContextInfo = NULL;
GUID guidContextID = GUID_NULL;

hres
=::CoGetObjectContext(IID_IObjectContextInfo,(void**)&pOb
jectContextInfo);
ASSERT(pObjectContextInfo != NULL);//not a configured
component?

hres = pObjectContextInfo->GetContextId(&guidContextID);
pObjectContextInfo->Release( );

USES_CONVERSION;
WCHAR pwsGUID[150];
::StringFromGUID2(guidContextID,pwsGUID,150);
TRACE("The object is in context with ID
%s",W2A(pwsGUID));
```

Note that only COM+-configured components should call `CoGetObjectContext( )`. When a nonconfigured component calls `CoGetObjectContext( )`, the call will fail with the return value of `E_NOINTERFACE`, and the returned interface pointer will be set to `NULL`. The assertion check in Example 2-1 tests for that condition.

One more point regarding the context object: the context object and its interfaces are private to the specific context they represent and should not be shared with or passed to objects in other contexts. Doing so may introduce hard-to-detect bugs and nondeterministic behavior of object deactivation and destruction, and it may affect transaction semantics and accuracy.

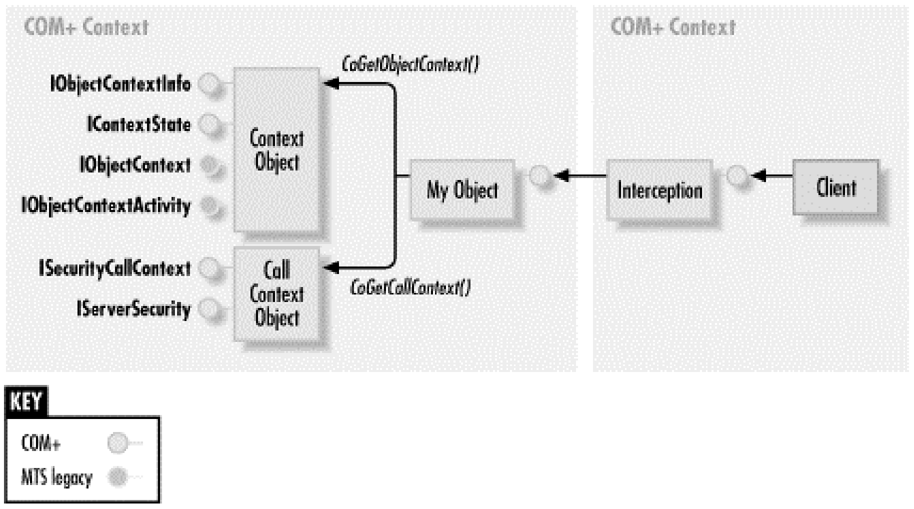
2.4 The Call Object

In addition to providing a context object to represent the context of an object, COM+ creates a transient object called the *call object* each time that object is called. The transient call object represents the current call in progress. Objects can access their call object by calling `CoGetCallContext( )` (see Figure 2-6). The `CoGetCallContext( )` signature is defined as:

```
HRESULT CoGetCallContext(REFIID riid, void**
ppInterface);
```

The call object only exists as long as a call from a client is in progress, and it is destroyed by COM+ after the called method returns. You should not cache a pointer to the call object as a member variable of your object because that pointer will be invalid once the method that saved it returns. Furthermore, if your object is doing work in the background—that is, no method call from the client is currently in progress—it will not have access to a call object. If you try to access a call object while a call is not in progress, `CoGetCallContext( )` will fail and return the error code `RPC_E_CALL_COMPLETE`. You can, however, still access the context object, which exists as long as the context exists, and whose pointer can be cached by the objects associated with it.
The call object exposes two interfaces used to obtain information about the call security settings. These interfaces, discussed in Chapter 7, are `ISecurityCallContext` and `IServerSecurity`.

Figure 2-6. When a method call is in progress, a COM+ object has access to the call object



2.5 Cross-Context Manual Marshaling

Cross-context call interception via marshaling is how COM+ provides its component services to your object. A client in a different context cannot access your object directly, even if it has a direct raw pointer to it. Intercepting the call and performing the right service switches requires a proxy and a stub in between. Otherwise, the object executes in the client context, possibly in an ill-suited runtime environment. If the client gets the pointer to your object in one of the following ways:

- CoCreating the object
- Querying an object the client already has for additional interfaces
- Receiving the pointer as a method parameter on a COM interface

Then COM+ will, under the hood, put interceptors (proxys and stubs) in place, to make sure all calls into the object are marshaled. If the client does anything else to obtain the interface pointer, such as retrieve it from a global variable or a static member variable shared among all clients, you have to marshal the pointer manually yourself. Dealing with pooled objects is another situation requiring manual marshaling, as you will see in the next chapter.
Classic COM requires that all cross-apartment calls be marshaled, even when the call is in the same process, to ensure threading model compatibility. The classic COM mechanisms for manually marshaling interface pointers across apartment boundaries have been made context-aware. They are what you should use to marshal interface pointers manually across context boundaries with COM+.
Generally, these mechanisms rely on the `CoMarshalInterface( )` and `CoUnmarshalInterface( )` functions. When you need to manually marshal an interface pointer from Context A to Context B, you would serialize the interface pointer into a stream in Context A using `CoMarshalInterface( )`, and get it out of the stream using `CoUnmarshalInterface( )` in Context B. This sequence would manually set up proxies in Context B for accessing the object. You can also use the `CoMarshalInterThreadInterfaceInStream( )` and `CoGetInterfaceAndReleaseStream( )` helper methods to automate some of the steps required when using just `CoMarshalInterface( )` and `CoUnmarshalInterface( )`.

2.5.1 The Global Interface Table

The preferred way to manually marshal interface pointers between contexts is by using the *global interface table* (GIT). Every process has one globally accessible table used for manually marshaling interface pointers. *Globally accessible* means accessible from every context and every apartment in the process. An interface pointer is checked into the GIT in one context. Then you get back an identifying cookie (a number), which is context-neutral and can be passed freely between clients across context boundaries, placed in global variable or class members, etc. Any client, at any context in the process, can access the GIT and use the cookie to get a properly marshaled interface pointer for its context. The GIT is only useful in cross-context marshaling in the same process and has no role in cross-process marshaling.

The GIT saves you the agony of programming directly against `CoMarshalInterface( )` or its helper functions, and more importantly, it overcomes a serious limitation of the `CoMarshalInterface( )` function. Using `CoMarshalInterface( )`, you can unmarshal an interface pointer just once for every `CoMarshalInterface( )` call. Using the GIT, you can check an interface pointer into the GIT once and check out interface pointers multiple times.

The GIT supports the `IGlobalInterfaceTable` interface, which is defined as:

```
interface IGlobalInterfaceTable : IUnknown
{
  HRESULT RegisterInterfaceInGlobal([in]IUnknown *pUnk,
                                    [in]REFIID  riid,
                                    [out]DWORD
*pdwCookie);
  HRESULT RevokeInterfaceFromGlobal([in]DWORD  dwCookie);
  HRESULT GetInterfaceFromGlobal([in]DWORD  dwCookie,
                                 [in]REFIID  riid,\
                                 [out]void**
ppInterface);
}
```

You can create the GIT with the class ID of `CLSID_StdGlobalInterfaceTable`.

`RegisterInterfaceInGlobal( )` is used to check an interface pointer into the GIT from within one context and to get back the identifying cookie. `GetInterfaceFromGlobal( )` is used to get a properly marshaled interface pointer at any other context using the cookie. `RevokeInterfaceFromGlobal( )` is used to remove the interface pointer from the GIT. Example 2-2 shows how to use the `IGlobalInterfaceTable` interface to manually marshal a pointer of type `IMyInterface` from Context A to Context B, or any other context in the process, using the GIT and a global variable.

Example 2-2. Manually marshaling a pointer using the GIT

```
//In context A:
HRESULT hres = S_OK;
extern DWORD dwCookie;//A global variable accessible in
any context
IMyInterface* pMyInterface = NULL;

/* Some code to initialize pMyInterface, by creating an
object that supports it*/

//Now, you want to make this object accessible from other
contexts.
dwCookie = 0;

//Create the GIT
IGlobalInterfaceTable* pGlobalInterfaceTable = NULL;
hres =
::CoCreateInstance(CLSID_StdGlobalInterfaceTable,NULL,

CLSCTX_INPROC_SERVER,IID_IGlobalInterfaceTable,

(void**)&pGlobalInterfaceTable);

//Register the interface in the GIT
hres = pGlobalInterfaceTable -
>RegisterInterfaceInGlobal(pMyInterface,

IID_IMyInterface,

&dwCookie);

pGlobalInterfaceTable->Release( );//Don't need the GIT
/////////////////////////////////////////////////////////
////////////////////////
//In context B:
IMyInterface* pMyInterface = NULL;
IGlobalInterfaceTable* pGlobalInterfaceTable = NULL;

hres =
::CoCreateInstance(CLSID_StdGlobalInterfaceTable,NULL,

CLSCTX_INPROC_SERVER,IID_IGlobalInterfaceTable,

(void**)&pGlobalInterfaceTable);

//Get the interface from the GIT
hres = pGlobalInterfaceTable-
>GetInterfaceFromGlobal(dwCookie,

IID_IGlobalInterfaceTable,

(void**)&pMyInterface);
```

```
pGlobalInterfaceTable->Release( );

/* code that uses pMyInterface */

pMyInterface->Release( );

////////////////////////////////////////////////////////
/////////////////////////
//Don't forget to revoke from the GIT when you are done
or before shutting down

IGlobalInterfaceTable* pGlobalInterfaceTable = NULL;

//You can use a cached pointer to the GIT or re-create
it:
hres =
::CoCreateInstance(CLSID_StdGlobalInterfaceTable,NULL,

CLSCTX_INPROC_SERVER,IID_IGlobalInterfaceTable,

(void**)&pGlobalInterfaceTable);

hres = pGlobalInterfaceTable-
>RevokeInterfaceFromGlobal(dwCookie);
pGlobalInterfaceTable->Release( );
```

The GIT increments the reference count of the interface pointer when it is registered. As a result, the client that registered the interface pointer can actually let go of its own copy of the interface pointer, and the object would not be destroyed. When you revoke the object from the GIT, the GIT releases its copy. When the process shuts down gracefully, if you forget to revoke your interfaces, the GIT revokes all the objects it still has, allowing them to be released. The GIT will `AddRef( )` an interface pointer that is returned from a call to `GetInterfaceFromGlobal( )`. A client should call a matching `Release( )` for every `GetInterfaceFromGlobal( )` called. Any client in the process can revoke a registered interface pointer. However, I recommend as a convention that the client who registered the object should be the one revoking it.

2.5.2 The GIT Wrapper Class

Using the raw global interface table has a few drawbacks. The resulting code is somewhat cumbersome and the `IGlobalInterfaceTable` method names are too long. In addition, the methods are not type safe because they require you to cast to and from a `void*` pointer. Previously, I saw a need for writing a simple C++ wrapper class that compensates for the raw usage drawbacks. The wrapper class provides better method names and

type safety, and because the class ID for the GIT is standard, its constructor creates the global interface table and its destructor releases it.
The wrapper class is called `CGlobalInterfaceTable` and is defined as:

```
template <class Itf,const IID* piid>
class CGlobalInterfaceTable
{
public:
   CGlobalInterfaceTable(  );
   ~CGlobalInterfaceTable(  );
   HRESULT Register(Itf* pInterface,DWORD *pdwCookie);
   HRESULT Revoke(DWORD dwCookie);
   HRESULT GetInterface(DWORD dwCookie,Itf**
ppInterface);

protected:
   IGlobalInterfaceTable* m_pGlobalInterfaceTable;

private://prevent misuse
   CGlobalInterfaceTable(const CGlobalInterfaceTable&);
   void operator =(const CGlobalInterfaceTable&);
};
```

By defining the GIT helper macro:

```
#define GIT(Itf) CGlobalInterfaceTable<Itf,&IID_##Itf>
```

You get automatic type safety because the compiler enforces the match between the interface ID and the interface pointer used.
Using the wrapper class is trivial. Here is the code required to retrieve an interface pointer from the table, for example:

```
IMyInterface* pMyInterface = NULL;
GIT(IMyInterface) git;
git.GetInterface(dwCookie,&pMyInterface);
```

Compare this code to Example 2-2. Using the wrapper class results in concise, elegant, and type-safe code. The GIT wrapper class is included as part of the source code available with this book.

2.6 Summary

This chapter introduced the COM+ context concept: a mechanism for providing component services. By intercepting client calls and performing additional processing, COM+ can ensure that the object has just the runtime environment it requires.
As stated at the beginning of this chapter, you usually do not need to interact with COM context or be aware that they exist. But understanding this abstract concept helps demystify the way COM+ services operate. Context and call interception is an extensible mechanism. As time goes by, new services can be added this way

without affecting existing applications. When a client creates instances of your old component in the new environment, COM+ silently does its context compatibility in the background, and your existing component never knows that new services are available.

Chapter 3. COM+ Instance Management

A few years ago, the dominant programming model and design pattern was the client/server model. COM and DCOM were predominant component technologies, and all was well. Then came the Internet revolution. Almost overnight, a new paradigm emerged—the *multitier architecture* . Scalability is perhaps the single most important driving force behind the move from classic two-tier client/server to multitier applications. Today, being able to handle a very large number of clients is necessary for survival. The classic two-tier model simply does not scale well from a few dozen clients to tens of thousands of clients hammering on your system at peak load. The two-tier model of dedicating one server object per client quickly causes critical resources to dwindle under such loads. Allocating resources such as a database connection, a system handle, or a worker thread to each client is unrealistic. The middle tier was introduced precisely because you could no longer map client objects directly to your data processing objects. The middle tier allows pooling of resources, such as database connections, hardware objects, or communication ports. The middle tier also allows you to activate your objects just when they are required and release them as soon as possible.

COM+ provides you with two elegant and user-friendly instance management services that you can use to build scalability into your system design from day one: object pooling and Just-in-Time Activation (JITA).

This chapter first defines the problems you face when designing a modern distributed system; it then explains COM+ strategies for managing objects that compose it.

3.1 Client Types

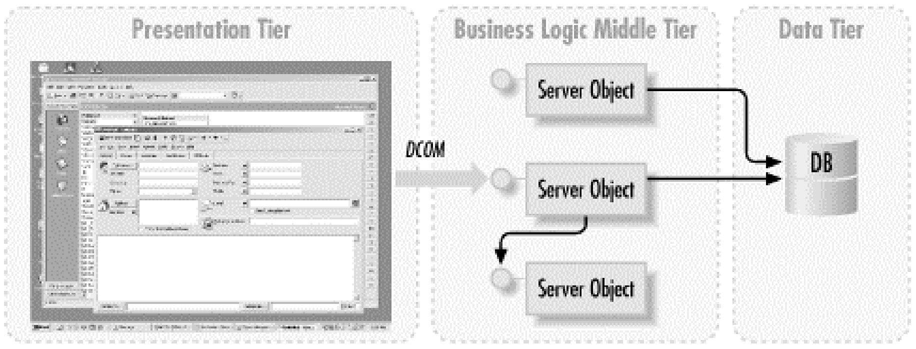
A distributed system, by its very nature, implies that its clients are not on the same machine as the objects providing the services. In every distributed system, there are typically two kinds of clients: *rich clients* and *Internet clients*. The rich client typically shares the same local area network, called the Intranet. (A rich client can also be called an *intranet client*.) In most cases, no firewalls between the rich client and the application exist, so the rich client can invoke binary calls on components in the application. The Internet client connects to your application typically by using a web browser, but more of the other options, such as hand-held PDAs and cellular phones, are possible as well. The Internet client is located outside of your local area network and can reside anywhere on the Internet. In

most cases, a firewall exists between the Internet client and your application.
Most applications have a mixture of rich and Internet clients. Some systems had only rich clients until they were opened to the Internet. Other systems were designed primarily for the Internet, but had to support rich clients—perhaps for application management, back-office operations, or other specific needs. In any case, when you design an application, you should plan to support both kinds of clients. The two kinds differ not only in the way they connect to your application, but also in their pattern of interaction with it. Your design should be able to scale up to both kinds of clients and compensate for their differences. COM+ instance management services were developed to answer precisely that challenge.

3.1.1 Scaling Up with Rich Clients

A rich client's interaction with the server objects of a distributed application resembles that of the classic client/server application. The client connects to the server machine using a network protocol such as TCP/IP. Because the Intranet is considered a secure environment, it usually contains no firewalls and the client can connect directly to your server objects in the middle tier using DCOM (see Figure 3-1). The calling pattern to your application is as follows: create an object, use it, and eventually release it. The rich client usually presents to the user a *rich user interface*. The word "rich" in this context means that the user interface contains and executes binary code, processing information and rendering it to the user. The user interface is typically built with tools such as Visual Basic or Visual C++ with MFC. The new .NET Framework provides a new library of Windows Forms classes for building rich clients. (See Chapter 10). Even if the user accesses your application with a web browser, that browser may contain binary ActiveX controls. Intranet clients use rich user interfaces because they must usually provide a rich user experience. This experience supports more privileges and features for employees than are available to customers connecting to the same system via an Internet browser.

Figure 3-1. Rich client connecting to a multitier system

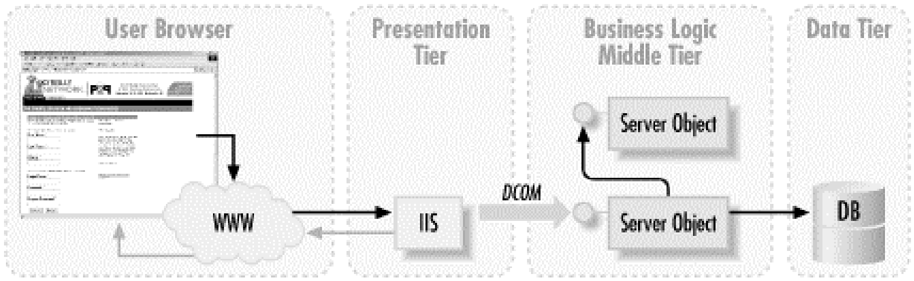


Consider your bank, for example. Most banks today provide easy access over the Internet for their customers, allowing simple operations such as viewing account balances and transferring funds between accounts. However, only bank tellers can perform operations such as opening or closing accounts and selecting various saving and investment plans. The next time you are in your bank, peek over the teller's screen. The teller probably uses a rich client user interface that does not look like the one you use when you log on to the Internet banking application offered by the bank. In a typical system, there are significantly fewer rich clients than Internet clients (as there are fewer bank tellers than bank customers). The overhead of creating a few server-side objects, allocating resources, and doing the cleanup for each client is not a scalability limitation. What really impedes scalability is the potential that rich client applications have for holding onto objects for long periods of time, while actually using the object in only a fraction of that time.
It is possible that when an Intranet application is started, it instantiates all the objects it needs and releases them only at shutdown, in an attempt to achieve better performance and responsiveness to the user. If your design calls for allocating an object for each client, you will tie up crucial limited resources for long periods and eventually run out of resources.

3.1.2 Scaling Up with Internet Clients

When users access your system over the Internet, they actually use a web browser to connect to an Internet web server (such as the Microsoft Internet Information Server, IIS). The browser generates a service request as part of the HTTP stream. The web server creates objects required for handling the client request, and when it finishes processing the request, it releases the objects (see Figure 3-2). It is important to emphasize that the Internet client connection is stateless; no object references are maintained outside the scope of individual requests. The client is usually a *thin* user interface, another name for an interface that consists of simple HTML rendered by a web browser. The browser's main job is to send the user's requests to the server and display the web server's reply. Although some scripts sent by the web server, such as Dynamic HTML (DHTML), may execute on the client side, such client-side logic is used primarily to format the information on the user's screen and has nothing to do with server-side objects.

Figure 3-2. Internet client connecting to a multitier system



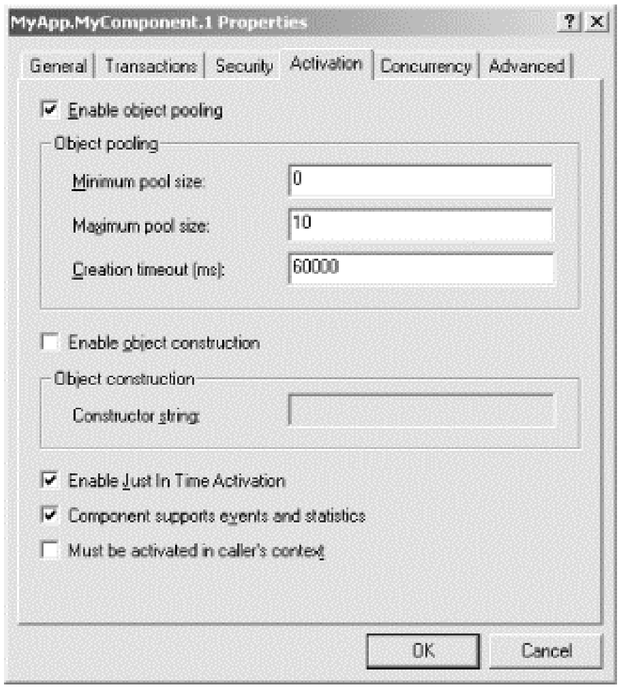
Depending on how widely your system is used, you could have a huge number of clients asking for service at any given moment. The length of time the web server holds the objects for an individual client request is usually not a scalability limitation. However, because there are so many Internet clients, scalability is limited by the overhead for each client request: creating objects, initializing them, allocating expensive resources such as database connections, setting up proxies, doing cross-machine or process calls, and doing cleanup. This problem is the opposite of the scalability problem for rich clients. Systems that use an ineffective approach of allocating objects per client request simply cannot handle a large number of clients. At periods of peak demand, the service appears to be unavailable or has irritatingly slow response time.

3.2 Instance Management and Scaling

Being smart about the way you allocate your objects to clients is the key to scalability in a modern distributed system. Simple algorithms can be used to govern when and how expensive objects that have access to scarce resources will actually service a client request. In distributed-systems terminology, these algorithms and heuristics are called *instance management*. COM+ refers to instance management as *activation*.

COM+ provides every configured component with access to ready-made instance management services. Every COM+ component has on its properties page an Activation tab that lets you control the way objects are created and accessed (see Figure 3-3). You can use COM+'s two instance management services, object pooling and JITA, individually, or combine them in a very powerful way. Neither technique is a COM+ innovation. What is new about COM+ is the ease with which you can take advantage of the service. That ease allows you to focus your development efforts on the domain problem at hand, not on the writing of instance management plumbing.

Figure 3-3. The COM+ component's Activation tab



[✓] Enable object pooling

Object pooling

Minimum pool size:	0
Maximum pool size:	10
Creation timeout (ms):	60000

[ ] Enable object construction

Object construction

Constructor string:

[✓] Enable Just In Time Activation

[✓] Component supports events and statistics

[ ] Must be activated in caller's context

3.3 Object Pooling

The idea behind object pooling is just as the name implies: COM+ can maintain a pool of objects that are already created and ready to serve clients. The pool is created per object type; different objects types have separate pools. You can configure each component type pool by setting the pool parameters on the component's properties Activation tab (as shown in Figure 3-3). With object pooling, for each object in the pool, you pay the cost of creating the object only once and reuse it with many clients. The same object instance is recycled repeatedly for as long as the containing application runs. The object's constructor and destructor are each called only once. Object pooling is an instance management technique designed to deal with the interaction pattern of Internet clients—numerous clients creating objects for every request, not holding references on the objects, but releasing their object references as soon as the request processing is done. Object pooling is useful when instantiating the object is costly or when you need to pool access to scant resources. Object pooling is most appropriate when the object initialization is generic enough to not require client-specific

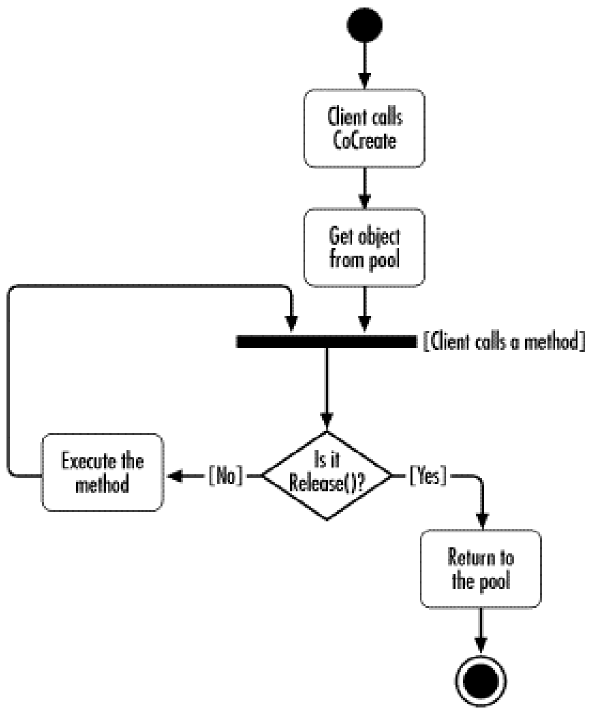
parameters. When using object pooling, you should always strive to perform in the object's constructor as much as possible of the time-consuming work that is the same for all clients, such as acquiring connections (OLEDB, ADO, ODBC), running initialization scripts, initializing external devices, creating file handles, and fetching initialization data from files or across a network. Avoid using object pooling if constructing a new object is not a time-consuming operation because the use of a pool requires a fixed overhead for pool management every time the client creates or releases an object.
Any COM+ application, whether a server or a library application, can host object pools. In the case of a server application, the scope of the pool is the machine. If you install proxies to that application on other machines, the scope of the pool can be the local network. In contrast, if the application is a library application, then a pool of objects is created for each client process that loads the library application. As a result, two clients in different processes will end up using two distinct pools. If you would like to have just one pool of objects, configure your application to be a server application.

3.3.1 Pooled Object Life Cycle

When a client issues a request to create a component instance and that component is configured to use object pooling, instead of creating the object, COM+ first checks to see if an available object is in the pool. If an object is available, COM+ returns that object to client. If there is no available object in the pool and the pool has not yet reached its maximum configured size, COM+ creates a new object and hands it back to the creating client. In any case, once a client gets a reference to the object, COM+ stays out of the way. In every respect except one, the client's interaction with the object is the same as if it were a nonpooled object. The exception occurs when the client calls the final release on the object (when the reference count goes down to zero). Instead of releasing the object, COM+ returns it to the pool. Figure 3-4 describes this life cycle graphically in a UML activity diagram.[1]

[1] If you are not familiar with UML activities diagrams, read *UML Distilled* by Fowler and Scott (Addison Wesley, 1997). Chapter 9 in that book contains a detailed explanation and an example.

Figure 3-4. A pooled object life cycle



If the client chooses to hold onto the pooled object for a long time, it is allowed to do so. Object pooling is designed to minimize the cost of creating an object, not the cost of using it.

3.3.2 Configuring Pool Parameters

To use object pooling for a given component, you should first enable it by selecting the "Enable object pooling" checkbox on component's Activation tab. The checkbox allows you to enable or disable object pooling. The two other parameters let you control the pool size and the object creation timeout. The minimum pool size determines how many objects COM+ should keep in the pool, even when no clients want an object. When an application that is configured to contain pools of objects is first launched, COM+ creates a number of objects for each pool equal to the specified minimum pool size for the application. If the minimum pool size is zero, COM+ doesn't create any objects until the first client request comes in. Minimum pool size is used to mitigate sudden spikes in demand by having a cache of ready-to-use, initialized objects. The minimum pool size must be less than the maximum pool size, and the Component Services Explorer enforces this condition.
The maximum pool size configuration is used to control the total number of objects that can be created, not just how many objects the pool can contain. For example, suppose you configure the pool to have a minimum size of zero and a maximum of four. When the first creation request comes in, COM+ simply creates an object and hands it over to the client. If a second request comes in and the first object is still tied up by the first client, COM+ creates a new

object and hands it over to the second client. The same is true for the third and fourth clients. However, when a fifth request comes along, four objects are already created and the pool has reached its maximum potential size, even though it is empty. Once you reach that limit and all objects are in use, further clients requests for objects are blocked until an object is returned to the pool. At that time, COM+ hands it over to the waiting client. If, on the other hand, the client waited for the duration specified in the timeout field, the client is unblocked and `CoCreateInstance( )` returns the error code `CO_E_ACTIVATIONFAILED_TIMEOUT` (not `E_TIMEOUT`, as documented in the COM+ section of the MSDN). COM+ maintains a queue for each pool of waiting clients to handle the situation in which more than one client is blocked while waiting for an object to become available. COM+ services the clients in the queue on a first-come, first-served basis as objects are returned to the pool. A creation timeout of zero causes all client calls to fail, regardless of the state of the pool and availability of objects.

If the pool contains more objects than the configured minimum size, COM+ periodically cleans the pool and destroys the surplus objects. There is no documentation of when or how COM+ decides to do the cleanup.

Deciding on the minimum and maximum pool size configuration depends largely on the nature of your application and the work performed by your objects. For example, the pool size can be affected by:

- Expected system load highs and lows
- Performance profiling done on your product to optimize the usage of resources
- Various parameters captured during installation, such as user preferences and memory size
- The number of licenses your customer has paid for; you can set the pool size to that number and have an easy-to-manage licensing mechanism

In general, when configuring your pool size, try to balance available resources. You usually need to trade memory used to maintain a pool of a certain size and the pool management overhead in exchange for faster client access and use of objects.

3.3.3 Pooled Object Design Requirements

When you want to pool instances of your component, you must adhere to certain requirements and constraints. COM+ implements object pooling by aggregating your object in a COM+ supplied wrapper. The aggregating wrapper's implementation of `AddRef( )` and `Release( )` manage the reference count and return the object to the pool when the client has released its reference. Your component must therefore support aggregation to be able to use object pooling. When you import a COM component into a COM+ application, COM+ verifies that your component supports aggregation. If it does not, COM+ disables object pooling in the Component Services Explorer. If you implement your object using ATL, make sure your code does not contain the ATL macro `DECLARE_NOT_AGGREGATABLE( )`, as this macro prevents your object from being aggregated. By default, the Visual C++ 6.0 ATL Wizard inserts this macro into your component's header file when generating MTS components. You must remove this macro to enable object pooling (it is safe to do so—there are no side effects in COM+).

Another design point to pay attention to is your pooled object's threading model. A pooled object should have no thread affinity of any sort—it should make no assumption about the identity of the thread it executes on, or use thread local storage, because the execution thread can be different each time the object is pulled from the pool to serve a client. The pooled object therefore cannot use the *single-threaded apartment* model (STA) because STA objects always require execution on the same thread. When you import a component to a COM+ application, if the component's threading model is marked as apartment (STA), COM+ disables object pooling for that component. A pooled object can only use the free *multithreaded apartment* model (MTA), the *both* model, or the *neutral threaded apartment* model (NTA, covered in Chapter 5). If performance is important to you, you may want to base your pooled component's threading model on your clients' threading model. If your clients are predominantly STA-based, mark your component as `Both` so that it can be loaded directly in the client's STA. If your clients are predominantly MTA based, mark your component as either `Free` or `Both` (the `Both` model also allows direct use by STA clients). If your clients are of no particular apartment designation, mark your component as `Neutral`. For most practical purposes, the neutral-threading model should be the most flexible and performance-oriented model. Table 3-1 summarizes these decisions.

Table 3-1. Pooled object threading model

Clients threading model	Recommended pooled object threading model
No particular model	NTA
STA	Both
MTA	Both/MTA
Both	Both
NTA	NTA

Deciding not to use STA has two important consequences:

- Pooled objects cannot display a user interface because all user interfaces require the STA message loop.
- You cannot develop pooled objects using Visual Basic 6.0 because all COM components developed in Version 6 are STA based and use thread local storage. The next version of Visual Basic, called Visual Basic.NET, allows you to develop multithreaded components.

3.3.4 Object Pooling and Context

When a pooled object is placed in the pool, it does not have any context. It is in stasis—frozen and waiting for the next client activation request. When it is brought out of the pool, COM+ uses its usual context activation logic to decide in which context to place the object—in its creator's context (if the two are compatible) or in its own new context. From the object's perspective, it is always placed in a new context; different from the one it had the last time it was activated. Objects often require context-specific initialization, such as retrieving interface pointers or fine-tuning security. Object pooling only saves you the cost of reconstructing a new object and initializing it to generic state. Each time an object is activated, you must still do a context-specific initialization, and you benefit from using object pooling only if the context-specific initialization time is short compared to that of the object's constructor. But when context-specific initialization is used, how does the object know it has been placed in a new context? How does it object know when it has been returned to the pool? It knows by implementing the `IObjectControl` interface, defined as:

```
interface IObjectControl : IUnknown
{
   HRESULT Activate( );
   void    Deactivate( );
   BOOL    CanBePooled( );
};
```

COM+ automatically calls the `IObjectControl` methods at the appropriate times. Clients of your object don't ever need to call these methods.

COM+ calls the `Activate( )` method each time the object is pulled from the pool to serve a client—just after it is placed in the execution context, but before the actual call from the client. You should put context-specific initialization in the `Activate( )` method. `Activate( )` is your pooled object's wakeup call—it tells it when it is about to start serving a new client. When using `Activate( )`, you should ensure that you have no leftovers in your object state (data members) from previous calls, or from a state that was modified from interaction with previous clients. Your object should be indistinguishable from a newly created object. The state should appear as if the object's constructor was just called.

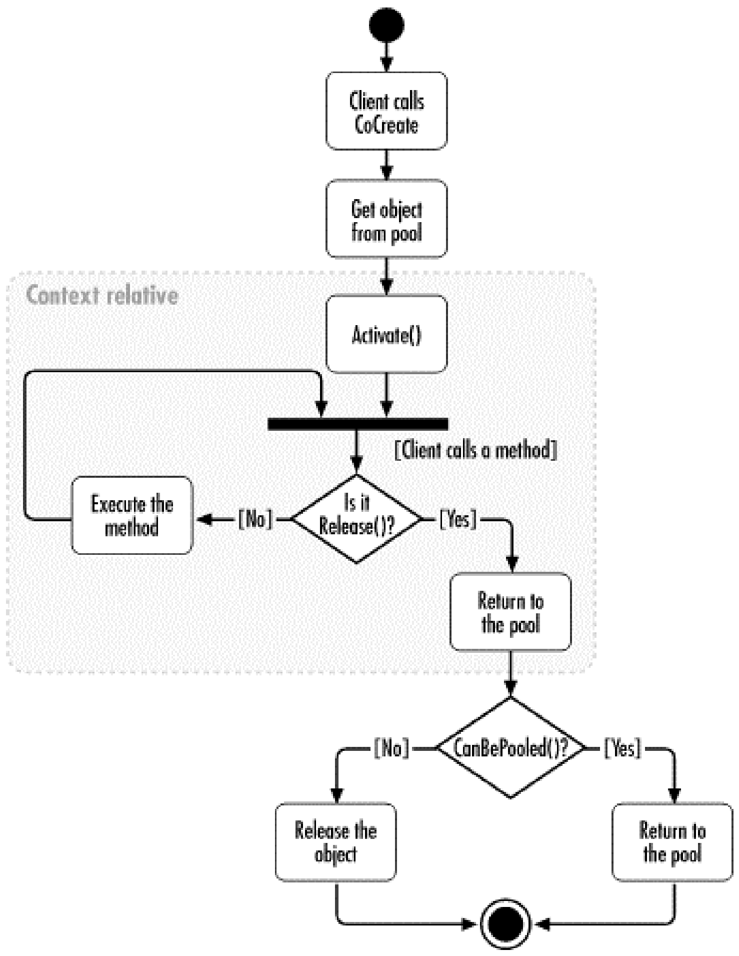
COM+ calls `Deactivate( )` after the client releases the object, but before leaving the context. You should put any context-specific cleanup code in `Deactivate( )`.

When object pooling is enabled, after calling the `Deactivate( )` method, COM+ invokes the `CanBePooled( )` method to let your object decide whether it wants to be recycled. This is your object's opportunity to override the configured object pooling setting at runtime. If your object returns `FALSE` from `CanBePooled( )`, the object is released and not returned to the pool. Usually, you can return `FALSE` when you cannot initialize the object's state to that of a brand-new object, because of an inconsistency or error, or if you want to have runtime fine tuning of the pool size and the number of objects in it. In the most cases, your implementation of `CanBePooled( )` should be one line: `return TRUE;`, and you should use the Component Services Explorer to administer the pool.

Implementing `IObjectControl` is not required for a pooled object. If you choose not to implement it and you enable object pooling, your object is always returned to the pool after the client calls `Release( )` on it.

Figure 3-5 emphasizes the calling sequence on a pooled object that supports `IObjectControl`. It shows when COM+ calls the methods of `IObjectControl` and when the object is part of a COM+ context.

Figure 3-5. The life cycle of a pooled object using IObjectControl



Finally, IObjectControl has two abnormalities worth mentioning: first, the interface contains two methods that do not return HRESULT, the required returned value according to the COM standard of any COM interface. IObjectControl's second abnormality is that only COM+ can invoke its methods. The interface is not accessible to the object's clients or to the object itself. If a client queries for the IObjectControl interface, QueryInterface( ) returns E_NOINTERFACE.

3.4 Just-in-Time Activation

Object pooling is a great instance management service, but what should you do when you deal with rich clients who can hold onto object references for long periods of time? It is one thing if the rich clients make intensive use of the object, but as you saw earlier, they actually maintain the reference on the object to improve performance on their side, and may actually call methods on the object for only a fraction of that time. From the object's perspective, it must still hold onto its resources because a call may

come through at any moment. Object pooling is of little benefit, since it saves you the cost of creating the object, not the cost of maintaining it while tied up with a client. Clearly, another tactic is required to handle greedy Intranet clients.
COM+ provides another instance management technique called *Just-in-Time Activation* (JITA) that allows you to dedicate an object per client only while a call is in progress. JITA is most useful when instantiating the object is not a costly operation compared with the expensive or scarce resources the object holds onto. It is especially useful if the object holds onto them for long periods.

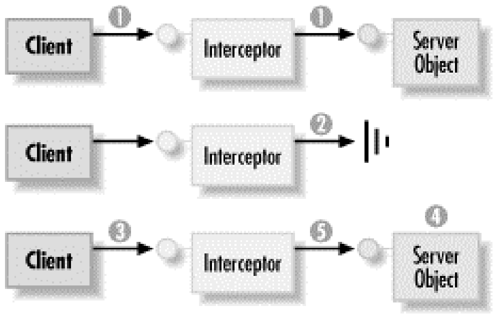
3.4.1 How JITA Works

JITA intercepts the call from the client to the object, activates the object just when the client issues a method call, and then destroys the object as soon as the method returns. As a result, the client must never have a direct reference to the object. As explained in Chapter 2, if the client is in a different context than the object, the client actually holds a pointer to a proxy and the proxy interacts with a stub. The COM+ proxy and stub perform the JITA interception, and together they constitute a single logical entity. Let's call this entity the *interceptor*. To guarantee that there is always an interceptor between the client and the object, component instances configured to use JITA are always placed in their own context, regardless of potential compatibility with their creator. Figure 3-6 shows how this interception works:

1. The interceptor calls the object's method on behalf of the client.
2. When the method call returns, if the object indicates that it can be deactivated, the interceptor releases the object and notes to itself that it no longer has the object. Meanwhile, the client continues to hold a reference to a proxy and does not know its object was released.
3. When the client makes another call, the interceptor notes that it is not connected to an object.
4. The interceptor creates a new object.
5. The interceptor delegates the call to the new object.

When the client releases the object, only the interceptor needs to be destroyed because the object was already released.

Figure 3-6. The interceptor handles the method calls in JITA by creating the object as it is needed and disposing of it between calls



3.4.2 Benefits of Using JITA

JITA is beneficial because you can now release the expensive resources the object occupies long before the client releases the object. By that same token, acquisition of the resources is postponed until a client actually needs them. Remember that activating and destroying the object repeatedly on the object side, without tearing down the connection to the client (with its client side proxy) is much cheaper than normally creating and releasing the object. Another side effect of JITA is that it improves overall reliability. Imagine the case of a client that crashed or simply forgot to release an object. When using JITA, the object and the resources it holds are released independently of unreliable or undisciplined clients.

3.4.3 Using JITA

You can configure any COM+ component to use JITA. On the Activation tab of the component's properties page (see Figure 3-3), you can check the "Enable Just In Time Activation" checkbox to enable JITA for your component. In fact, when you use the Component Installation Wizard to add a new component, it enables JITA for the new component by default.
However, COM+ cannot arbitrarily kill your object just because the method has returned. What if an object is not ready to be deactivated? What if it needs to perform additional activities to bring itself to a consistent state, and can only then be destroyed? An object that wants to get the most out of JITA is required to do two things: first, it should be state-aware. Second, it should tell COM+ when the object can be deactivated. Mind you, a JITA object does not need to be stateless. In fact, if it were truly stateless, there would be no need for JITA in the first place. The object has to proactively manage its state, much like a transactional object, as discussed in the next chapter. Ideally, a JITA object should be activated at the beginning of every method call and deactivated after the call. If you intend to signal to COM+ to deactivate your object only after a particular method returns or when a special

event has occurred, the client may hold onto the object between the calls for long periods of time and significantly hamper scalability.
So, if a JITA object is to be activated just before every method call and deactivated immediately after each call, then it needs to do two things: at the beginning of each call, the object should initialize its state from values saved in durable storage. At the end of the call, it should return its state to the storage. Commonly used durable storage options include databases and the Windows filesystem.

> Although a JITA object can store its state in nondurable storage, namely in-memory, I recommend not doing so for two reasons. First, if the JITA object participates in transactions (discussed in the next chapter), the storage has to be durable. Second, memory storage ties the object to a particular machine and precludes multimachine load balancing.

Not all of the state of an object can be saved by value. For example, if the state includes interface pointers to other objects, the object should release those objects and re-create them on the next activation. A database connection from a connections pool is another example of a state that cannot be stored. An object should return the connection to the pool before returning from a method call and grab a new connection from the pool upon activation.
Using JITA has one important implication for interface design—every method call must include a parameter to identify the object of which the method is a member. The object uses that parameter to retrieve its state from the durable storage, and not the state of another instance of the same type. Examples for such parameters include the account number for bank account objects and the order number for objects representing store orders. Example 3-1 shows a method on a JITA object that accepts a parameter of type PARAM (a pseudotype invented for this example) used to identify the object:

```
STDMETHODIMP CMyClass::MyMethod(PARAM objectIdentifier)
```

The object then uses the identifier to retrieve its state and save the state back at the end of the method call.
JITA clearly offers you a tradeoff between performance (the overhead of reconstructing the object state on each method call) and scalability (holding onto the state and its resources). No definitive rules describe when and to what extent you should trade performance for scalability. You may need to profile your system, and ultimately redesign some objects to use JITA and some not to use JITA. Nevertheless, JITA is a powerful instance management technique available with one click of your mouse.
JITA also lets COM+ know when it is allowed to deactivate the object. You have already seen that each JITA object must reside in a context separate from that of its caller. Each context has a

context object associated with it, as explained in the previous chapter. Each context object has a value in it called the *done bit* , which, as the name implies, is a one-bit Boolean flag. Whenever a context is initialized, and an object is placed in it, the done bit is set to zero (`FALSE`).

A JITA object lets COM+ know that it is ready to be destroyed by setting the done bit on the context object to `TRUE`. The object interceptor checks the done bit every time a method returns control to it. If the done bit is set to `TRUE`, the interceptor releases the JITA object. Because each COM+ context maps to a single context object, a JITA object always resides in its own private, dedicated context. If more than one object were in the context, any one could set the done bit to `TRUE`, and the interceptor might deactivate the wrong object.

You can set the value of the done bit either programmatically or administratively. You can set the done bit programmatically in two ways, and both require accessing an interface exposed by the context object. The recommended way to set the done bit for a JITA object is to use the interface `IContextState`, an interface that Microsoft fine tuned to support JITA objects. Its definition is as follows:

```
enum tagTransactionVote
{
   TxCommit= 0,
   TxAbort = TxCommit + 1
}TransactionVote;

interface IContextState : IUnknown
{
   HRESULT SetDeactivateOnReturn([in] BOOL bDeactivate);
   HRESULT GetDeactivateOnReturn([out]BOOL*
pbDeactivate);
   HRESULT SetMyTransactionVote ([in]TransactionVote
txVote);
   HRESULT GetMyTransactionVote ([out]TransactionVote*
ptxVote);
}
```

`IContextState` defines methods for setting the done bit and retrieving its current value. `IContextState` is also used in transaction voting, discussed in the next chapter. You can obtain `IContextState` by using the call `CoGetObjectContext( )`; you can call the `IContextState` method `SetDeactivateOnReturn( )` to set the done bit, as shown in Example 3-1.

Example 3-1. Using IContextState to tell COM+ to deactivate the object

```
STDMETHODIMP CMyClass::MyMethod(PARAM objectIdentifier)
{
  GetState(objectIdentifier);
  DoWork(  );
  SaveState(objectIdentifier);
  //Let COM+ deactivate the object once the method
returns
  HRESULT hres = S_OK;
  IContextState* pContextState = NULL;
  hres =
::CoGetObjectContext(IID_IContextState,(void**)&pContextS
tate);
  ASSERT(pContextState != NULL)//Will be NULL if not
imported to the COM+ Explorer

  hres = pContextState->SetDeactivateOnReturn(TRUE);
  ASSERT(hres != CONTEXT_E_NOJIT)//will return
CONTEXT_E_NOJIT if JITA was not
                                 //enabled for this
object
  pContextState->Release(  );
}
```

Another way of setting the done bit uses the `IObjectContext` interface. You can obtain this interface by using `CoGetObjectContext( )` and calling its `SetComplete( )` method. However, `IObjectContext` is a legacy interface from MTS, and using it to deactivate the object can have transaction voting side effects, discussed in the next chapter. A COM+ JITA object should use `IContextState`.

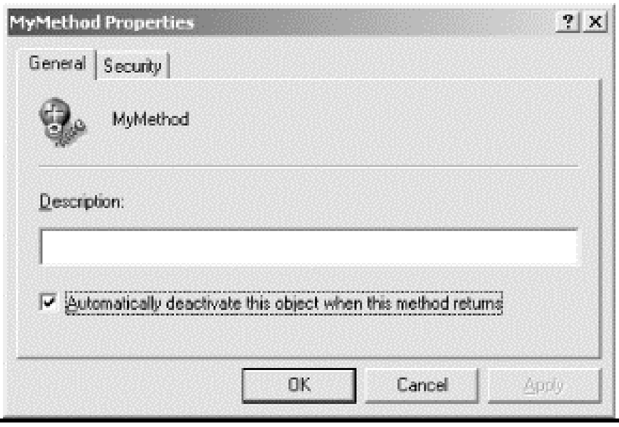
Deactivating a JITA Object Developed in VB 6.0

If you use Visual Basic 6.0 to develop your JITA object, you must access `IObjectContext` first, and then query it for `IContextState` to flag the object for deactivation:

```
Dim objectContext As ObjectContext
Dim contextState As IContextState
Set objectContext = GetObjectContext

'QueryInterface for IContextState:
Set contextState = objectContext
contextState.SetDeactivateOnReturn (True)
```

Programmatic control over when COM+ should deactivate your object gives you ultimate control over when deactivation occurs. When using JITA, however, you are more likely to want to deactivate your object each time a method returns. COM+ provides you with an administrative way to instruct it to always deactivate the object upon method return. When JITA is enabled for a component, you can search in the Component Services Explorer for the method level, display the method properties page, select the General tab, and check "Automatically deactivate this object when this method returns" (see Figure 3-7).

Figure 3-7. The method's General tab



The "Automatically deactivate" setting is done at the method level, not the interface level. This setting potentially leaves the object with some methods that do not deactivate the object on return (especially if they do not acquire expensive resources) and some that do. However, for consistency's sake, you should set all interface methods and all component interfaces in a uniform fashion.

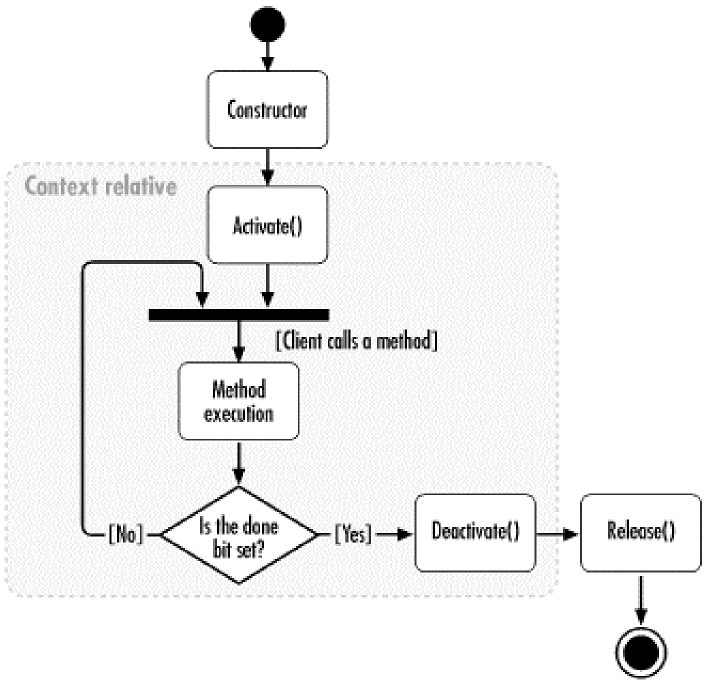
Even when you administratively configure a method to deactivate the object when it returns, you can still override this configuration programmatically at runtime by calling `IContextState::SetDeactivateOnReturn(FALSE)`. COM+ only uses the administrative setting when you do not make a programmatic call yourself to set the context object's done bit.

3.4.4 JITA and IObjectControl

Your JITA object can choose to implement the `IObjectControl` interface. COM+ queries for the interface and calls `IObjectControl::Activate( )` each time a new instance of your component is created and placed in the COM+ context after the object constructor is called, but before the actual method is called. By letting the object know when it enters a context, COM+ allows the object to perform context-specific initialization in `Activate( )`, such as passing a reference to the object to another object, caching interface pointers to other COM+ objects (such as `IContextState`), or performing programmatic security checks (see Chapter 7).

If you set the done bit to TRUE, after the method has returned (but before the object destructor is called) COM+ calls IObjectControl::Deactivate( ). You should put your context-specific cleanup code, such as releasing cached interface pointers, in Deactivate( ). After calling Deactivate( ), COM+ destroys (releases) the object.
If the JITA object is not configured to use object pooling, then COM+ never calls IObjectControl::CanBePooled( ). However, you still must implement all the methods of a COM interface your object support. Just return TRUE from CanBePooled( ), which makes your object support pooling (you may still want to configure it to support pooling in the future).
Figure 3-8 shows the life cycle of a JITA object that implements IObjectControl. A JITA object that supports IObjectControl is notified by COM+ when the object is placed in a new context and also just before the object leaves the context and is destroyed.

Figure 3-8. Life cycle of a JITA object that implements IObjectControl



Here are a few more important points about JITA objects and IObjectControl:

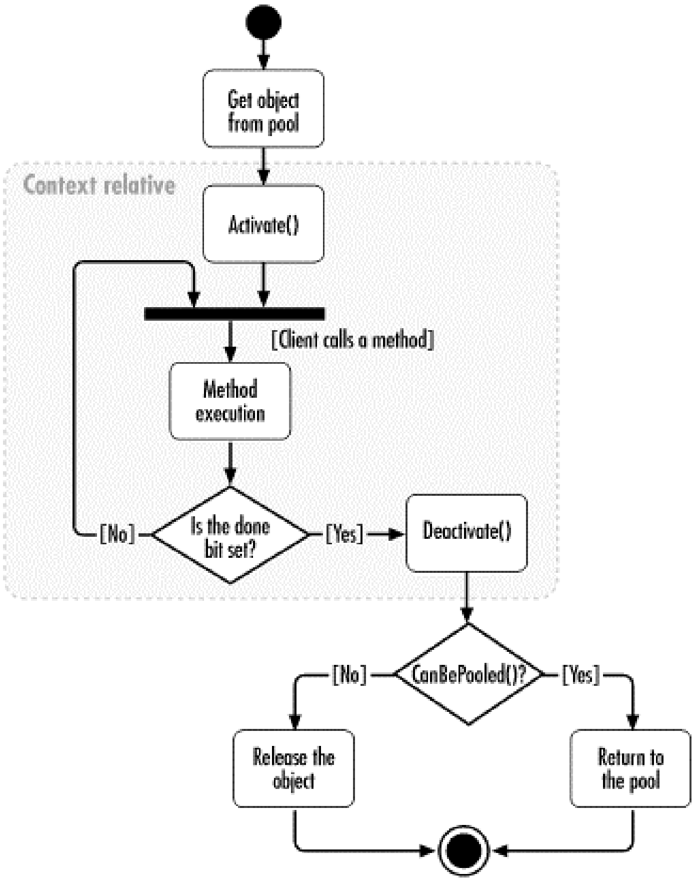
- If you return anything except S_OK from IObjectControl::Activate( ), perhaps out of failure to initialize a context-specific state, the client gets the HRESULT of CO_E_INITIALIZATIONFAILED as a return value from the method it wanted to call.

- Merely enabling JITA and implementing `IObjectControl` will not get your object deactivated after every method call—you must either configure the method administratively or set the done bit programmatically. If you do not want to use JITA, but wish to know when you enter a context in order to do context-specific initialization, you can enable JITA support and implement `IObjectControl`.
- Even though implementing `IObjectControl` is optional, I strongly recommend that you implement it when you use JITA because it makes managing your object's life cycle much easier.

3.5 Combining JITA with Object Pooling

The two instance management techniques provided by COM+ are not mutually exclusive. JITA and object pooling can be combined in a very powerful way. Using both object pooling and JITA on the same component is useful in situations when object initialization is both generic (not client specific) and expensive. Thus, using just JITA would not make sense; when you have no control over the length of time, the object's client keeps its reference to the object, so you would realize marginal gain from object pooling. When you configure your object to use both, instead of creating and releasing the object on each method call, COM+ grabs an object from the pool and returns the object to the pool after the method completes its execution. The JITA aspects are still maintained because the object instance will be torn away from its client. The pool will also be used on every method call, not just on `CoCreate` and `Release` calls from the client. Implementing `IObjectControl` is optional, but I strongly recommend it. As always, a call to `IObjectControl::Activate( )` marks entry to a context, and a call to `IObjectControl::Deactivate( )` marks an exit. COM+ calls `IObjectControl::CanBePooled( )` after every `Deactivate( )`, letting the object decide whether it wants to be recycled or destroyed. This life cycle is shown in Figure 3-9. When you configure your component to support both JITA and object pooling, COM+ deactivates the object every time the done bit is set and returns it to the pool instead of releasing it. New method calls are served by recycled objects from the pool, not with new instances.

Figure 3-9. The life cycle of a component using JITA and object pooling



Objects now can maintain state between calls because they are not destroyed, but rather returned to the pool. The truth is, when you use JITA and object pooling together, your object still cannot maintain a client-specific state between invocations; Once the object is back in the pool, it could very well be retrieved to serve a different client than the previous one. A JITA object can maintain just the generic part of the state and benefit from going through that initialization only once.

When a pooled object is configured to use JITA, the semantics of the maximum pool size value actually sets the total number of objects that COM+ is forced to create to serve active client calls, not the total number of connections to clients. The number of connections (the number of clients holding references to proxies) may be a much larger number because many clients may not be engaged in calling a method on objects.

Configuring a COM+ component to be a singleton is an interesting example of what you can do when combining JITA with object pooling. A singleton is a component with only one instance. All clients share the same singleton—the clients are often not even aware that there is just one instance of the class.[2] For example, suppose you have a configured component used to control a single resource, such as a hardware device or a communication port. To make sure that all clients get the same object, you can configure your component to use JITA and object pooling, with minimum and maximum pool sizes set to one. Having a pool size of exactly one ensures that at any given moment, exactly one object (a singleton) is associated with a resource. Using JITA ensures that once the object has finished servicing one client, it can serve another, even if the current client has not released its reference to it. The singleton is also the only case of a JITA object that can maintain full state between method calls, since you can be certain that the same object is called to serve all clients. However, before you start using a singleton, make sure that its disadvantages (a single point of failure, a performance hot spot, a bottleneck, and an inability to scale to large number or clients) are not relevant in your design and that it is a valid modeling of an entity in your application domain.

[2] See *Design Patterns—Elements of Reusable Object-Oriented Software*, by Gamma, et al. (Addison Wesley, 1995), p. 127.

3.6 Object Constructor String

COM+ allows you to pass a construction parameter to new instances of your component. This instance activation service has nothing to do with application scalability, JITA, or object pooling, and is nothing more than a neat service.
On your component's Properties page, there is a properties group named "Object construction" on the Activation tab. Once you enable this service (by checking the "Enable object construction" checkbox), you can specify a string in free form. Every instance of your component has access to this one string (you cannot specify a string per instance). Because calls to `CoCreateInstance( )` or `CreateObject( )` do not accept initialization parameters, you have to work to gain access to the constructor string.
The first thing you need to do (besides enable the service) is have your component implement an interface called `IObjectConstruct`, defined as:

```
interface IObjectConstruct : IUnknown
{
   HRESULT Construct([in]IDispatch* pConstructionObj);
};
```

If you enable object construction but do not implement the interface, all client attempts to create a new instance of your component will fail, showing the error code `E_NOINTERFACE`. They will fail because COM+ will refuse to hand over to the client an object that could not be initialized properly. `IObjectConstruct` has only one method, `Construct( )`, which COM+ uses to pass in a pointer to another interface called `IObjectConstructString`, defined as:

```
interface IObjectConstructString : IDispatch
{
   [id(1),propget] HRESULT ConstructString([out, retval]
BSTR* pVal);
};
```

COM+ calls your object's implementation of `IObjectConstruct::Construct( )` to deliver the string only once, immediately after calling the object constructor. Note that COM+ passes the construction string to your object before the call to `IObjectControl::Activate( )`, since the initialization parameter should provide generic, rather than context-specific, information. Example 3-2 shows how to use the constructor string object passed into `IObjectConstruct::Construct( )` to access your component's configured constructor string.

Example 3-2. Implementing IObjectConstruct::Construct( ) and accessing your component's configured constructor string

```
// IObjectConstruct::Construct(  )
STDMETHODIMP CMyComponent::Construct(IDispatch *
pConstructionObj)
{
   HRESULT hres = S_OK;
   BSTR    bstrConstruct;

   IObjectConstructString* pString = NULL;
   hres = pConstructionObj-
>QueryInterface(IID_IObjectConstructString,

(void**)&pString);

   hres = pString->get_ConstructString(&bstrConstruct);
   pString->Release(  );

   //Use bstrConstruct

   return S_OK;
}
```

Note that empty strings may be valid parameters and that your object should be written to handle an empty string.

However, why go through a somewhat odd mechanism of retrieving the string from a dedicated interface, rather than passing `IObjectConstruct::Construct( )` a `BSTR` directly? The answer is that in the future, COM+ may allow you to pass other kinds of parameters for construction, such as numbers, data structures, or maybe even interface pointers. The COM+ designers wanted to put

in place a generic mechanism that could extend to handling more than just strings.
You can use a construction string to specify parameters common to all components, but whose value is deployment specific, such as:

- Log filename and location. The COM+ logbook, presented in Appendix A, uses the constructor string to do just that.
- Application or component configuration filename and location.
- If your component holds a generic ODBC connection, you can specify a DSN file name—referencing a file containing information about the database to be used by this component—instead of either passing it in as a method parameter or hardcoding it.

3.7 COM+ Instance Management Pitfalls

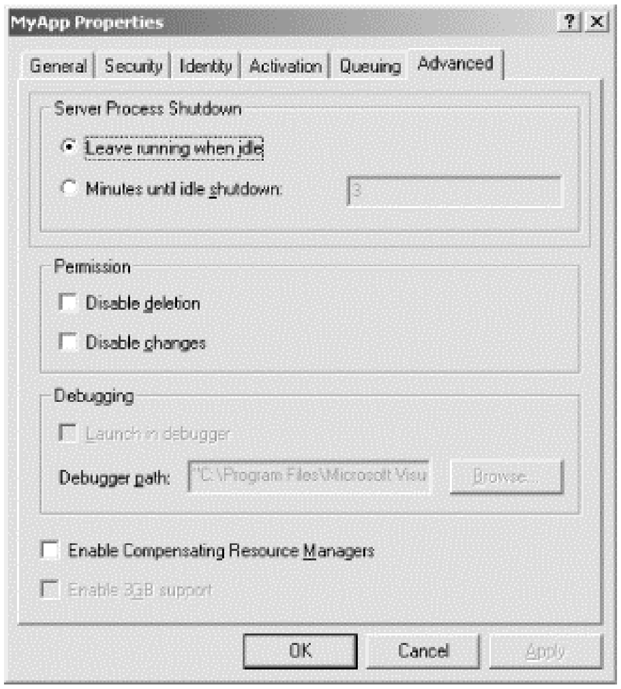
COM+ instance management and object activation have a few minor pitfalls and limitations you should be aware of to make the best use of what COM+ has to offer. This section also discusses a feature of the Component Services Explorer that will help you profile your application and keep track of your object instances.

3.7.1 Idle Time Management

Under classic COM, a process hosting COM objects would be left running as long as clients with active references to objects are in that process. Once the last client releases its reference on the last object in that process, COM would shut down the hosting process. This policy clearly conflicts with COM+ object pooling—the idea is to keep objects alive, even if they do not serve any clients. COM+ allows you to configure your server application's idle time management on the Advanced tab of the application's properties page (see Figure 3-10). The Advanced tab has a properties group called Server Process Shutdown. If your application contains pools of objects, you can leave the hosting process running when the application is idle—that is, when the application is not servicing clients and all objects are in the pool. However, your objects continue to occupy resources as long as the process is running, and if the client activation requests are few and far between, this may not be a good tradeoff.
Alternatively, you can specify how long you want to keep the application idle by providing any number between 0 and 999 minutes. You should decide on the exact value based on your clients' calling pattern and optimize the overall activation overhead and resource consumption. For example, if you expect the interval

between clients' activations of pooled objects to be 10 minutes, you should configure the application to be left idle at least that long, plus a certain safety factor (20 percent for example). In this case, you would set the idle timeout to 12 minutes. If you set the timeout to 0, you will get the classic COM behavior. Setting the timeout to 0 is especially useful during debugging because as long as the application is running, you cannot rebuild the component DLL; you cannot rebuild it because the application process has that DLL loaded and locked. Usually, when you discover a defect during a debug session, you should fix it, rebuild the component, and retest. By setting the timeout to 0, you can rebuild immediately. By default, after creating a new COM+ application, the application is configured to shut down after 3 minutes of idle time.

Figure 3-10. Consider leaving an application hosting object pools running even when idle



Server Process Shutdown
[✓ Leave running when idle]
Minutes until idle shutdown: 3

Permission
Disable deletion
Disable changes

Debugging
Launch in debugger
Debugger path: "C:\Program Files\Microsoft Visu

Enable Compensating Resource Managers
Enable 3GB support

3.7.2 Too Large a Minimum Pool Size

If you set a component to have an object pool with a minimum size greater than zero, then when the application containing the component is launched, COM+ creates the minimum size number of objects and puts them in the pool. The first activation request for any object (pooled or not) from that application may take a long time to complete if you have a too large a minimum pool size. Objects from your application may end up paying the pool initialization price, resulting in slow response time to their clients. To mitigate this problem, consider starting your application explicitly, either manually from the Component Services Explorer or programmatically by programming with the COM+ catalog, as explained in Chapter 6.

3.7.3 Requesting a JITA Object in the Caller's Context

The Component Services Explorer lets you require that a component always be activated in its creator's context by checking the "Must be activated in caller's[3] context" checkbox on the Activation tab in the component properties page (see Figure 3-3). If the creating client were in another context, the activation call would fail with the error code `CO_E_ATTEMPT_TO_CREATE_OUTSIDE_CLIENT_CONTEXT`.

[3] The name is inaccurate—it should be "Must be activated in creator's context."

You can use this setting only when you are sure that the creating client will not be in another process and will have configuration settings close enough to allow the component instance to share its context. This setting is available as an advanced optimization service for cases when the calling client makes short, frequent calls to the component and the overhead of cross-context marshaling gets in your way.
As you saw before, a JITA instance must have its own context so it can have its own done bit to set and an in interceptor between it and the client. The two settings, "Enable Just In Time Activation" and "Must be activated in the caller's context," are mutually exclusive, yet the Component Services Explorer gladly lets you set a component to use both settings. Beware of configuring a JITA object to always be activated in its caller's context because this configuration causes all activation requests to fail.

3.7.4 Failing to Release Pooled Object Data Members

When retrieved from the pool, a pooled object should be placed in a different context on each activation. As explained in the previous chapter, object references under COM+ are context-relative and must be marshaled between contexts. Your design of the pooled object may have it include interface pointers to other objects as data members. Those references are required for the pooled object to function properly. In fact, such references may be the very reason why you made it a pooled object, if creating the contained objects takes a long time to complete. Clearly, your object cannot create the contained objects and save them as data members because the data members would be invalidated on the next deactivation.
You can get around this problem in two ways. First, you can create the contained objects, register them in the Global Interface Table (GIT) (covered in the previous chapter), and save the identifying

GIT cookies as data members, rather than raw interface pointers. The pooled object should implement `IObjectControl`, and on every call to `IObjectControl::Activate( )`, it should get the objects out of the GIT and have a current-context safe copy of the data members. When COM+ calls `IObjectControl::Deactivate( )`, the object should release its local copy. When the pooled object is finally released, it should revoke the interface pointers from the GIT.
The second solution would use pooled objects as data members. On every call to `IObjectControl::Activate( )`, the pooled object should create (retrieve from their pools) all the helper objects it needs, and on calls to `IObjectControl::Deactivate( )`, it should release its local copies. Because the helper objects are pooled objects themselves, there should not be much of a penalty for creating them. The only thing you should remember is to configure the various pools to have enough objects in them. You can, of course, mix the two solutions (have some objects pooled and use the GIT on the rest). As always, you, as the application designer, are responsible for finding the right solution for your design and addressing the particular constraints of the domain problem at hand.

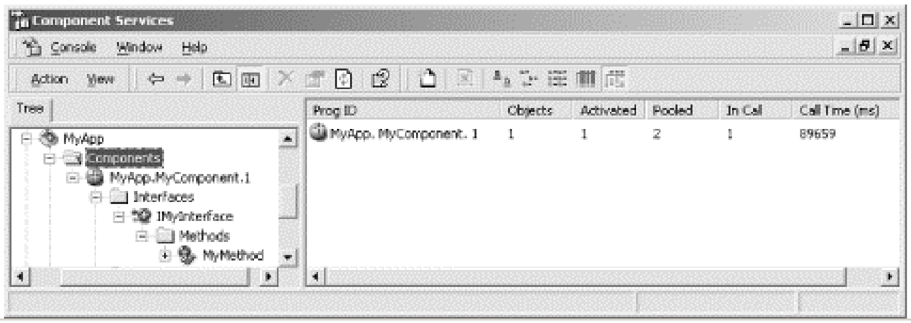
3.7.5 Pooled Objects and Aggregation

COM+ implements object pooling by aggregating your object and intercepting the activation calls from the client. By doing so, COM+ keeps track of your pooled object and manages its life cycle (returns it to the pool instead of releasing it and calls `IObjectControl` methods at appropriate times). As a result, your pooled object is discouraged from aggregating other COM/COM+ objects. Imagine, for example, that Object A is a pooled object, and it aggregates another pooled Object B. COM+ aggregates Object A and manages its activation recycling, but who would manage Object B's recycling? Because there is no way for the client to tell that a given object is pooled, it is better to be safe than sorry in this case. Even if you are certain that Object B is not a pooled object, there is no guarantee that it will not be configured to be a pooled object in the future. Avoid aggregation within a pooled object.

3.7.6 Tracking Instance Activity

When developing a configured component that takes advantage of COM+ instance management services, it is sometimes hard to keep track of exactly what is going on with instances of your component: how many are in the pool, how many are actually servicing clients, etc. Trying to gauge the various parameters, such as pool size and activation timeout, may require a lot of profiling of the average call time and the client's calling patterns. The Component Services Explorer provides you with crucial information to help you develop and fine tune your application. If you expand the component folder and select the Status View from the toolbar, COM+ displays various statistics on instances of each component in your application (see Figure 3-11). The status view columns description is in Table 3-2. You will find the status view helpful in almost all phases of development and deployment.

Figure 3-11. Select the component folder Status View to display various statistics on instances of your components



Prog ID	Objects	Activated	Pooled	In Call	Call Time (ms)
MyApp. MyComponent. 1	1	1	2	1	89659

Table 3-2. The Component folder status view columns, from left to right

Column	Description
Prog ID	The component identifying programmatic ID.
Objects	The total number of outstanding references to objects of this type. If you use JITA, this number is the number of clients that still hold a reference to an instance. The number may be much larger than all the other numbers.
Activated	The number of currently activated objects—objects that are in a context tied up with a client. If the object uses JITA and sets the done bit to `TRUE` after every call, then the number in the Activated column will be the same as the number in the In Call column.
Pooled	The total number of pooled objects created. This number includes both the objects in the pool and pooled objects outside the pool that services clients.
In Call	The number of objects currently executing method calls on behalf of clients. This number is always less than or equal to the Activated column because the objects can use JITA and deactivate themselves between calls.
Call Time	The average call time in milliseconds of all the calls, on all the methods, across all instances in the last 20 seconds. A call time is defined as the time it took the object to execute the call, and does not include object activation, the time spent marshaling the call across context, process, or machine boundary.

Collecting the statistics causes a small performance hit. COM+ only presents the status information on objects that are configured to provide it. Configure your component to support statistics on the component Activation tab by checking "Component supports events and statistics" (see Figure 3-3). By default, COM+ enables this support when you install a new configured component.

Chapter 4. COM+ Transactions

Consider the everyday operation of withdrawing cash from an automated teller machine (ATM), an operation you perform frequently. You access your account, specify the amount to withdraw, and then receive cash from the machine. Yet even an operation this mundane involves multiple machines (the ATM, the bank mainframe, and probably a few other machines) and multiple databases (an accounts database, a money transfer database, an audit database, and so on), each of which may also reside on a machine of its own. At the ATM itself, the withdrawal involves both a software user interface and mechanical devices such as the card reader, keypad, bill delivery mechanism, and receipt printer.

The difficulty in developing an ATM application lies in the fact that all of these steps can succeed or fail independently of the others. For example, suppose the ATM can't connect to the mainframe at the bank or for some reason cannot execute your request. Or, suppose there is a security problem (the wrong PIN code was entered) or the hardware fails (the ATM runs out of bills).

In addition, multiple users may access the bank's system simultaneously. Their access and the changes they make to the system must be isolated from one another. For example, while you are withdrawing money at the ATM, your spouse could be accessing the account online and a teller could be doing a balance check for a loan approval.

Nevertheless, both you and the bank expect either all the operations involved in accomplishing the request to succeed, or all the operations to fail. Partial success or partial failure of a banking transaction is simply not acceptable; you don't want the bank to deduct the money from the customer's account but not dispense the bills, or to dispense the bills but not deduct money from the account.

The expectation for an all-or-nothing series of operations characterizes many business scenarios. Enterprise-level services such as funds management, inventory management, reservation systems, and retail systems require an all-or-nothing series of operations. A logical operation (such as cash withdrawal) that complies with this requirement is called a *transaction*.

The fundamental problem in implementing a transactional system is that executing all the operations necessary to complete the transaction requires transitioning between intermediate *inconsistent system states*—states that cannot themselves be tolerated as valid outcomes of the transaction. For example, an inconsistent state would result if you were to deduct money from one account but not credit it to another in a simple transfer of funds between the two accounts. In essence, an inconsistent state is any system-state that

is the result of partial success or failure of the elements of one logical operation.
One approach to addressing the complex failure scenarios of a transaction is to add error-handling code to the business logic of your application. However, such an approach is impractical. A transaction can fail in numerous ways. In fact, the number of failure permutations is exponentially proportional to the number of objects and resources participating in the transaction. You are almost certain to miss some of the rare and hard-to-produce failure situations. Even if you manage to cover them all, what will you do when the system evolves—when the behavior of existing components changes and more components and resources are added, thereby multiplying the number of errors you have to deal with? The resulting code will be a fragile solution. Instead of adding business value to the components, you will spend most of your time writing error-handling code, performing testing and debugging, and trying to reproduce bizarre failure conditions. Additionally, the tons of error-handling code will introduce a serious performance penalty.
The proper solution is not to have the transaction error-handling logic in your code. Suppose the transaction could be abstracted enough that your components could focus on executing their business logic and let some other party monitor the transaction success or failure. That third party would also ensure that the system be kept in a consistent state and that the changes made to the system (in the case of a failed transaction) would be rolled back.
That solution is exactly the idea behind the COM+ transaction management service. COM+ simplifies the use of transactions in the enterprise environment. COM+ provides administrative configuration of transactional support for your components. COM+ enables auto-enlistment of resources participating in the transaction and supports managing and executing the transaction across machine boundaries. The COM+ transaction management service is based on the MTS transactions management model, with a few improvements and innovations.

4.1 Transaction Basics

Before we discuss COM+ transaction support, you need to understand the basics of transaction processing, the fundamental properties that every transaction must have, and some common transaction scenarios. If you are already familiar with the basic transaction concepts, feel free to skip directly to Section 4.4 later in this chapter.
Formally, a transaction is a set of potentiality complex operations that will all succeed or fail as one atomic operation. Transactions

are the foundation of electronic information processing, supporting almost every aspect of modern life.
Transactions were first introduced in the early 1960s by database vendors. Today, other resource products, such as messaging systems, support transactions as well. Traditionally, the application developer programmed against a complex *Transaction Processing Monitor* (TPM)—a third party that coordinated the execution of transactions across multiple databases and applications. The idea behind a TPM is simple: because any object participating in a transaction can fail and because the transaction cannot proceed without having all of them succeed, each object should be able to help determine success or failure of the entire transaction. This is called *voting* on the transaction's outcome. While a transaction is in progress, the system can be in an inconsistent state. When the transaction completes, however, it must leave the system in a consistent state—either the state it was in before the transaction executed or a new one.
Transactions are so crucial to the consistency of an information system that, in general, whenever you update a persistent storage (usually a database), you need to do it under the protection of a transaction. Another important transaction quality is its duration. Well-designed transactions are of short duration because the speed with which your application can process transactions has a major impact on its scalability and throughput. For example, imagine an online retail store. The store application should process customer orders as quickly as possible and manage every client's order in a separate transaction. The faster the transaction executes, the more customers per second the application can service (throughput) and the more prepared the application is to scale up to a higher number of customers.

4.2 Transaction Properties

Modern standards call for a transaction to be *atomic, consistent, isolated,* and *durable.* In transaction processing terminology, these properties are referred to as the *ACID* properties. When you design transactional components, you must adhere to the ACID requirements; they are not optional. As you will see, COM+ enforces them rigorously. Once you understand the ACID requirements and follow simple design guidelines, developing transactional components in COM+ becomes straightforward.

4.2.1 The Atomic Property

When a transaction completes, all the changes it made to the system's state must be made as if they were all one *atomic*

operation. The word *atom* comes from the Greek word *atomos,* meaning *indivisible*. The changes made to the system are made as if everything else in the universe stops, the changes are made, and then everything resumes. It is not possible to observe the system with only some of the changes.
A transaction is allowed to change the system state only if all the participating objects and resources execute their part successfully. Changing the system state by making the changes is called *committing* the transaction. If any object encounters an error executing its part, the transaction aborts and none of the changes is committed. This process is called *aborting* the transaction. Committing or aborting a transaction must be done as an atomic operation.
A transaction should not leave things to do in the background once it is done, since those operations violate atomicity. Every operation resulting from the transaction must be included in the transaction itself.
Because transactions are atomic, a client application becomes a lot easier to develop. The client does not have to manage partial failure of its request or have complex recovery logic. The client knows that the transaction either succeeded or failed as a whole. In case of failure, the client can choose to issue a new request (start a new transaction) or do something else, such as alert the user. The important thing is that the client does not have to recover the system.

4.2.2 The Consistent Property

A transaction must leave the system in a consistent state. Note that consistency is different from atomicity. Even if all changes are committed as one atomic operation, the transaction is required to guarantee that all those changes are consistent—that they make sense. The component developer is responsible for making sure the semantics of the operations are consistent. A transaction is required to transfer the system from one consistent state to another. Once a transaction commits, the system is in a new consistent state. In case of error, the transaction should abort and roll back the system from the current inconsistent and intermediate state to the initial consistent state.
Consistency contributes to simple client-side code as well. In case of failure, the client knows that the system is in a consistent state and can use its higher-level logic to decide the next step (or maybe none at all, since the system is in a consistent state).

4.2.3 The Isolated Property

While a transaction is in progress, it makes changes to the system state. *Isolation* means no other entity (transactional or not) is able to see the intermediate state of the system. The intermediate state shouldn't be seen outside of the transaction because it may be inconsistent. Even if it were consistent, the transaction could still abort and the changes could be rolled back. Isolation is crucial to overall system consistency. Suppose Transaction A allows Transaction B access to its intermediate state. Transaction A aborts, and Transaction B decides to commit. The problem is that Transaction B based its execution on a system state that was rolled back, and therefore Transaction B is left unknowingly inconsistent. Managing isolation is not trivial. The resources participating in a transaction must lock the data accessed by the transaction from all others and must synchronize access to that data when the transaction commits or aborts. The transaction monitoring party should detect and resolve deadlocks between transactions using timeouts or queues. A *deadlock* occurs when two transactions contend for resources the other one holds. COM+ resolves deadlocks between transactions by aborting the deadlocked transactions.

Theoretically, various degrees of transaction isolation are possible. In general, the more isolated the transactions, the more consistent their results are, but the lower the overall application throughput—the application's ability to process transactions as fast as it can. COM+ 1.0 transactions use the highest degree of isolation, called *serialization* . This term means that the results obtained from a set of concurrent transactions are identical to the results obtained by running each transaction serially. To achieve serialization, all the resources a transaction in process touches are locked from other transactions. If other transactions try to access those resources, they are blocked and cannot continue executing until the original transaction commits or aborts. The next version of COM+ (see Appendix B) allows configuring the isolation level of your transactions and trades consistency for throughput.

4.2.4 The Durable Property

If a transaction succeeds and commits, the changes it makes to the system state should persist in a *durable* storage, such as a filesystem, magnetic tapes, or optical storage. Transactions require commitment of their changes to a durable storage because at any moment the machine hosting the application could crash and its memory could be erased. If the changes to the system's state were in-memory changes, they would be lost and the system would be in an inconsistent state. The changes a transaction makes to the system state must persist even if the machine crashes immediately after the decision to commit the changes is made. The component's developer is required to store the new system state only in durable resources. The durable resource must be robust enough to

withstand a crash while trying to commit the changes. One way to achieve such robustness would be to manage log files to recover from the crash and complete the changes.
However, how resilient to catastrophic failure the resource really should be is an open question that depends on the nature and sensitivity of the data, your budget, available time, and available system administration staff. A durable system can be anything from a hard disk to a RAID disk system that has multiple mirror sites in places with no earthquakes.

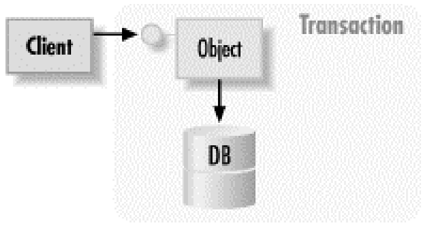
4.3 Transaction Scenarios

Applications differ greatly in their complexity and need for COM+ transactions support. To understand the COM+ transactions architecture and the needs it addresses, you should first examine a few generic transaction cases.

4.3.1 Single Object/Single Resource Transaction

Consider an application that comprises just one component instance, an object that processes a client's request and accesses a single resource (such as a database) that takes part in a transaction. This situation is depicted in Figure 4-1. The application (in this case, the object) has to inform the resource when a transaction is started. This act is called *enlisting* the resource in the transaction. The object starts making calls on the resource interfaces, making changes to its state. However, at this point the resource should only record (log) the changes and not actually perform them.
If the object encounters no errors when executing a client's request, then on completion it informs the resource that it should try to commit the changes. If the object encounters errors, it should instruct the resource to abort and roll back the changes. Even if the object wants to commit the transaction, any existing errors on the resource side might cause the transaction to abort.

Figure 4-1. Managing a transaction in a single object/single resource scenario



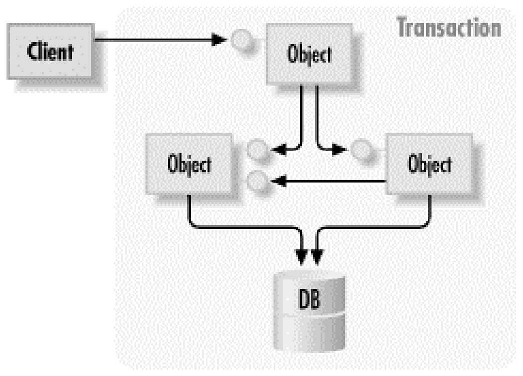
Note that only the application can request to commit the transaction, but either the application or the resource can abort it.

You can easily deal with a single object/single resource scenario on your own without relying on COM+ transactions by making explicit programmatic calls to enlist a resource in a transaction and instructing it to commit or roll back at the end of the transaction. Most resources support this sort of interaction out-of-the-box and expose simple functions, such as `BeginTransaction( )` and `EndTransaction(commit/abort)`.

4.3.2 Multiple Objects/Single Resource Transaction

Suppose you have multiple objects in your application, each of which requires access to the same resource to service a particular client request. Suppose your design calls for containing all the changes the objects make to the resource in the same transaction, to ensure consistency of these multiple changes (see Figure 4-2).

Figure 4-2. Multiple components with a single resource transaction



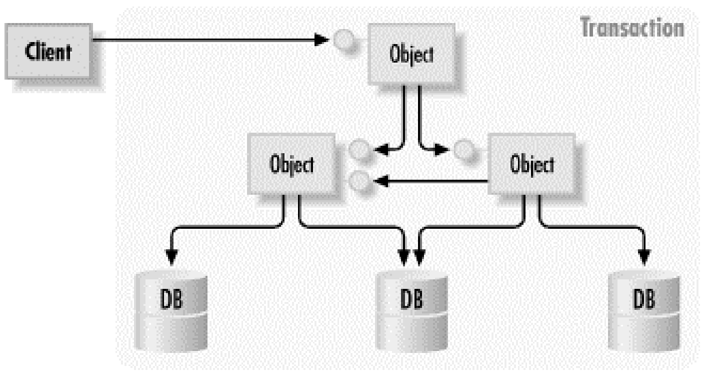
Unfortunately, things get much more complicated than in the previous scenario. The main problem is coordination. Since the resource should be enlisted in the transaction just once, who should be responsible for enlisting it? Should it be the first object that accesses it? Or maybe it should be the first object that is created? How would the objects know and coordinate this information? In addition, since the objects can all be on different machines, how would you propagate the transaction from one machine to the next? How would the objects know what transaction they are in? What should you do if one machine crashes while the other machines continue to execute the client request?
Each of the objects can encounter errors and abort the transaction, and they ask the resource to commit the changes only if they all succeed. The problem here is deciding which object is responsible for collecting the votes. How would an object know that a transaction is over? Who is responsible for notifying the resource of the voting result—that is, instructing the resource to try to commit or roll back the changes? What should the objects do with their own state (their data members)? If the resource is unable to commit the changes, the transaction must abort; in that case, the objects' state

reflects inconsistent system state. Who will inform the objects to purge their inconsistent state? How would the objects know what part of their state constitutes system inconsistency?
Fortunately, COM+ transactions support makes this scenario as easy to deal with as the previous one. COM+ takes care of enlisting the resource, propagating the transaction across machine boundaries, collecting the components' votes, and maintaining overall resource and object state consistency.

4.3.3 Multiple Objects/ Multiple Resources Transaction

An enterprise application often consists of multiple objects accessing multiple resources within the same transaction (see Figure 4-3).

Figure 4-3. An enterprise application comprising multiple components and resources



In addition to all the coordination challenges posed by the previous scenario, you now have to enlist all the resources just once in the transaction. Who keeps track of what resources are used? You definitely don't want to have that knowledge in your code because it could change. Who is responsible for informing the resources about the transaction outcome (the components' votes) and asking them to try to commit or abort? Since any one of the resources can refuse to commit the changes, how do you know about it and how would you instruct the other resources to roll back their changes? Your components and resources may all be on different machines, resulting in multiple points of failure. Transaction processing monitors (TPMs) have evolved to answer these challenges, but they require explicit calls from the application, which results in a cumbersome programming model.
Yet again, COM+ transactions support makes this situation as easy as the first one. Even in a distributed environment with multiple resources, your programming model is elegant and simple. It allows you to focus on your business logic while relying on COM+ to manage the transaction for you.

4.4 COM+ Transactions Architecture

COM+ is an advanced TPM that provides your components with easy-to-use administrative configuration for your transactional needs. COM+ encapsulates the underlying transaction monitoring and coordination required to manage a transaction. The COM+ transactions architecture defines a few basic concepts you need to understand to take advantage of COM+ transactions support: resource managers, the transaction root, the two-phase commit protocol, and the Distributed Transaction Coordinator (DTC).

4.4.1 Resource Managers

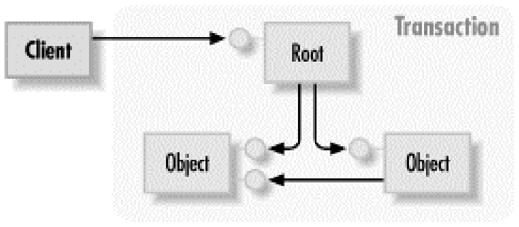
A resource (such as a database management system) that can participate in a COM+ transaction is called a *resource manager.* A resource manager knows how to conduct itself properly in the scope of a COM+ transaction—it records the changes done by your application's objects and will only commit the changes when told to do so. A resource manager knows how to discard the changes and revert to its previous state if it is told to roll back. A resource manager can *auto-enlist* in a transaction—the resource manager can detect it is being accessed by a transaction and enlist itself in it. Every COM+ transaction has a unique transaction ID (a GUID), created by COM+ at the beginning of the transaction. The resource manager keeps track of the transaction ID and will not enlist twice. Auto-enlisting means that your components are not required to explicitly enlist the resources needed for a transaction; therefore, they do not have to deal with the problem of multiple objects accessing the same resource, not knowing whether or not it is already enlisted in the transaction.
A resource manager must store its data in a durable storage to maintain the transaction durability. To maintain the transaction's isolation, a resource manager must lock all data (such as rows, tables, and queues) touched by the transaction, and allow only objects that take part in that transaction to access that data. Note that all the hard work required to manage a resource manager is hidden from your components. The burden is on the resource manager's shoulders, not yours.
A resource manager must vote on the transaction's result. Once the transaction is over, COM+ asks each participating resource manager, "If you were asked to commit the changes, could you?". A resource manager is represented by a system service that manages the resource, and your objects access the resource manager via a proxy.
Quite a few resources today comply with these requirements: first and foremost is Microsoft SQL Server (Versions 6.5 and above), but

other non-Microsoft databases, such as Oracle 8*i* and IBM DB2, are COM+ resource managers as well. A resource manager does not have to be a database; for example, Microsoft Message Queue (MSMQ) is a resource manager.

4.4.2 Transaction Root

When multiple objects take part in a transaction, one of them has to be the first to ask that a transaction be created to contain the operation (usually a client's request). That first object is called the transaction *root*. A given transaction has exactly one root (see Figure 4-4).

Figure 4-4. A transaction's root object



Designating an object as a transaction's root, or as an *internal* object, is done administratively. The component's developer configures it to either not take part in transactions; to require a transaction, (to join an existing transaction if one exists); or to start a new transaction if none exists. If the component starts a new transaction, then it becomes the root of that transaction. The developer can also configure the component to always start a new transaction—to always be the root of a new transaction.
Once a transaction is created, when Object A in Transaction T1 creates another object, Object B, according to B's configuration, it will:

- Be part of Transaction T1.
- Not be part of T1 or any other transaction. This may compromise isolation and consistency because B can perform operations that will persist even if T1 aborts. Also, B has no way of deciding whether T1 should abort in case B has an error.
- Start a new Transaction T2. In that case, Object B becomes the root of the new transaction. This option may also compromise isolation and consistency, as one transaction could commit and the other one could abort independently of the other.

Neither A nor B needs to actively do anything to decide on the transaction. COM+ checks the object's configuration and places it in the correct transaction automatically.

4.4.3 The Two-Phase Commit Protocol

COM+ uses a transaction management protocol called the *two-phase commit* to decide on a transaction result, commit changes to the system state, and enforce atomicity and consistency. The two-phase commit protocol enables COM+ to support transactions that involve multiple resources.

After the transaction's root starts a new transaction, COM+ stays out of the way. New objects may join the transaction, and every resource manager accessed automatically enlists itself with that transaction. The objects execute business logic and the resource managers record the changes made under the scope of the transaction. You already saw that all the application's objects in a transaction must vote during the transaction for whether the transaction should abort (if the objects had an error) or be allowed to commit (if the objects have done their work successfully). Again, abstaining from voting on the transaction's outcome is not an option for any object in the transaction. A transaction ends when the root object is released (or deactivated, when you're using JITA). At that point, COM+ steps back into the picture and checks the combined vote of the participating objects. If any object voted to abort, the transaction is terminated. All participating resource managers are instructed to roll back the changes made during the transaction.

If all the objects in the transaction vote to commit, the two-phase commit protocol starts. In the first phase, COM+ asks all the resource managers that took part in the transaction if they have any reservations in committing the changes recorded during the transaction. Note that COM+ is not instructing the resource managers to commit the changes. COM+ merely asks for their vote on the matter. At the end of the first phase, COM+ has the combined vote of the resource managers. The second phase of the protocol acts upon that combined vote. If all resource managers voted to commit the transaction in the first phase, then COM+ would instruct all of them to commit the changes. If even one of the resource managers said in phase one that it could not commit the changes, then in phase two, COM+ would instruct all the resource managers to roll back the changes made, thus aborting the transaction.

It is important to emphasize that a resource manager's vote that has no reservations about committing is special: it is an unbreakable promise. If a resource manager votes to commit a transaction, it means that it cannot fail if, in the second phase, COM+ instructs it to commit. The resource manager should verify before voting to commit that all the changes are consistent and legitimate. A resource manager never goes back on its vote. This is the basis for enabling transactions. The various resource manager vendors have gone to great lengths to implement this behavior exactly.

4.4.4 The Distributed Transaction Coordinator

As demonstrated in the transaction scenarios described previously, there is a clear need to coordinate a transaction in a distributed environment, to monitor the objects and resources in the transaction, and to manage the two-phase commit. Managing the interaction between the components (by collecting their votes) is done by COM+; managing the two-phase commit protocol is done by the *Distributed Transaction Coordinator* (DTC). The DTC is a system service tightly integrated with COM+. The DTC creates new transactions, propagates transactions across machines, collects resource managers' votes, and instructs resource managers to roll back or commit.

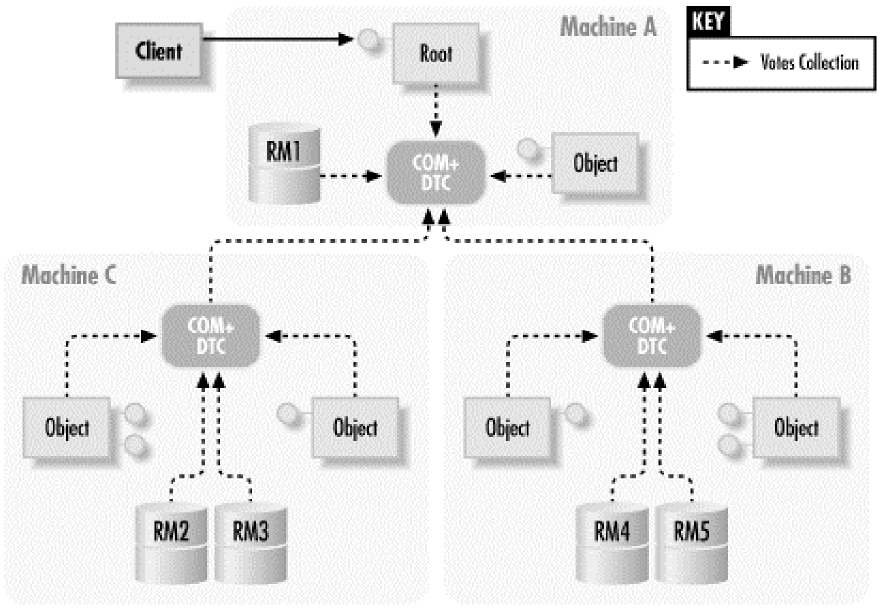
Every machine running COM+ has a DTC system service. When an object that is part of a transaction on Machine A tries to access another object or a resource on Machine B, it actually has a proxy to the remote object or resource. That proxy propagates the transaction ID to the object/resource stub on Machine B. The stub contacts the local DTC on Machine B, passing it the transaction ID and informing it to start managing that transaction on Machine B. Because the transaction ID gets propagated to Machine B, resource managers on Machine B can now auto-enlist with it.

When the transaction is done, COM+ examines the combined transaction vote of all participating objects. If the combined vote decides to abort the transaction, COM+ instructs all the participating resource managers on all participating machines to roll back their changes. If the combined objects' vote was to try to commit the transaction, then it is time to start the two-phase commit protocol. The DTC on the root machine collects the resource managers' votes on the root machine and contacts the DTC on every machine that took part in the transaction, instructing them to conduct the first phase on their machines (see Figure 4-5). The DTCs on the remote machines collect the resource managers' votes on their machines and forward the results back to the DTC on the root machine.

After the DTC on the root machine receives the results from all the remote DTCs, it has the combined resource managers' vote. If all of them voted to commit, then the DTC on the root machine again contacts all the DTCs on the remote machines, instructing them to conduct phase two on their respective machines and to commit the transaction. If, however, even one resource manager voted to abort the transaction, then the DTC on the root machine informs all the DTCs on the remote machines to conduct phase two on their respective machines and abort the transaction. Note that only the

DTC on the root machine has the combined vote of phase one, and only it can instruct the final abort or commit.

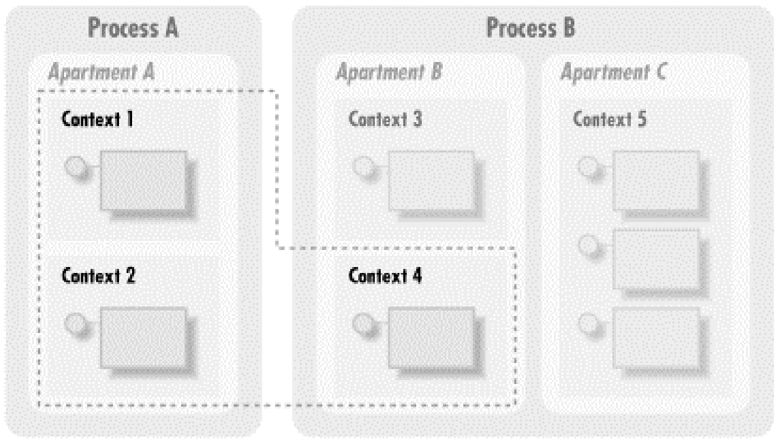
Figure 4-5. COM+ and the DTC manage a distributed transaction



4.4.5 Transactions and Context

A given transaction can contain objects from multiple contexts, apartments, processes, and machines (see Figure 4-6).

Figure 4-6. A transaction (whose scope is indicated by the dashed line) is unrelated to machine, process, apartment, and context



Each COM+ context belongs to no more than one transaction, and maybe none at all. COM+ dedicates a single bit in the context object (discussed in Chapter 2) for transaction voting. An object votes on a transaction's outcome (whether to proceed to phase one of the two-phase commit protocol or to abort) by setting the value of that bit. As a result, a transactional object must have its own

private context. Two transactional objects cannot share a context because they only have one bit to vote with. If two objects share a context and one of them wants to abort and the other wants to commit, then you would have a problem. Therefore, each COM+ object belongs to at most one transaction (because it belongs to exactly one context) and an object can only vote on the outcome of its own transaction. Collecting the object's vote is done by the context's interceptor when the object is released or deactivated.
The context object has more to do with the transaction than just holding the object's vote bit. Internally, each context object stores references to the transaction it belongs to, if any exist. The context object stores the transaction's ID and a pointer to the transaction object itself. Every transaction is represented by an interface called `ITransaction`, and the context object stores an `ITransaction*` pointer to the current transaction it belongs to. You can gain access to that information by accessing the context object and obtaining the `IObjectContextInfo` (first presented in Chapter 2), defined as:

```
interface IObjectContextInfo : IUnknown
{
   BOOL IsInTransaction( );
   HRESULT  GetTransaction(IUnknown** ppTransaction);
   HRESULT  GetTransactionId([out] GUID* pTransactionID);
   HRESULT  GetActivityId([out] GUID* pActivityID);
   HRESULT  GetContextId([out] GUID* pContextId);
};
```

The `GetTransactionId( )` method returns the transaction ID (a GUID). The `IsInTransaction( )` method returns `TRUE` if the context is included in a transaction. The `GetTransaction( )` method returns a pointer to the current transaction this context is part of, in the form of a `ITransaction*` interface pointer.
A full discussion of the `ITransaction` interface is beyond the scope of this chapter. It is used by resource managers to auto-enlist in a transaction and to vote during the two-phase commit protocol. Briefly, when the object accesses a resource manager, it does so via a proxy. The resource manager's proxy retrieves the transaction ID and the `ITransaction*` pointer from the context object and forwards them to the resource manager for auto-enlistment. The resource manger looks at the transaction ID. If it is already enlisted in that transaction, then it does nothing. However, if this is the first time the resource manager is accessed by that transaction, it uses the `ITransaction*` pointer to enlist.

4.4.6 COM+ Transactions Architecture Benefits

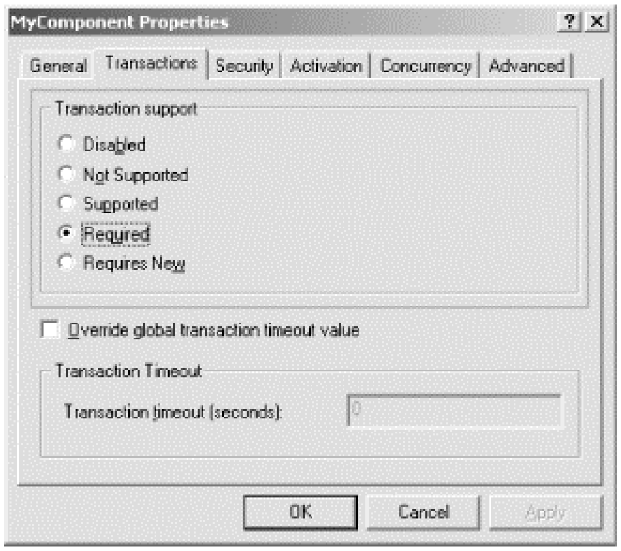
The benefits of COM+ transactions architecture were implied in the previous discussion of the architecture's elements. Now that you have the comprehensive picture, you can see that the main benefits are as follows:

- Auto-enlistment of resource managers saves you the trouble of making sure that resources are enlisted exactly once. Otherwise, components would be coupled to one another by having to coordinate who enlists what resource and when.
- An object and its client do not ever need to know what the other objects are doing, whether they require transactions, or what another object's vote is. COM+ places objects in transactions automatically, according to their configuration. COM+ collects the objects' votes and rollback changes. All an object has to do is vote.
- The programming model is simplified, robust, easier, and faster to implement.
- The COM+ transactions architecture decouples the components from specific TPM calls. There is nothing in the components' code that relates to the DTC or to transaction management.

4.5 Configuring Transactions

Now that you understand what transactions are and what they are good for and have reviewed the COM+ transaction architecture, it is time to put that knowledge into practice to build and configure transactional components in COM+.
You can use the Component Services Explorer to configure transaction support for your components. Every component has a Transactions tab on its properties page. The tab offers you five options for transaction support (see Figure 4-7): Disabled, Not Supported, Supported, Required, and Requires New. The settings let you control whether instances of your component take part in a transaction and if so, whether and when they should be the root of that transaction.

Figure 4-7. Configure transaction support for a component on the component's Transactions tab



Transaction support

- Disabled
- Not Supported
- Supported
- [✓ Required]
- Requires New

☐ Override global transaction timeout value

Transaction Timeout

Transaction timeout (seconds): 0

COM+ determines which transaction to place the object in when it creates the object. COM+ bases its decision on two factors: the transaction of the object's creator and the configured transaction support of the object (actually, for the component that the object is an instance of).
A COM+ object can belong to its creator's transaction, be a root of a new transaction, or not take part in a transaction. If the object is configured with transaction support Disabled or Not Supported, it will never be part of a transaction, regardless of whether its creator has a transaction or not. If the object is configured with Supported and its creator has a transaction, then COM+ places the object in its creator's transaction. If the creating object does not have a transaction, then the newly created object will not have a transaction. If the object is configured with transaction support set to Required, then COM+ puts it in its creator's transaction if the creating object has a transaction. If the creating object does not have a transaction and the object is configured to require a transaction, COM+ creates a new transaction for the object, making it the root of that new transaction. If the object is configured with transaction support set to Requires New, then COM+ creates a new transaction for it, making it the root of that new transaction, regardless whether its creator has a transaction or not. The COM+ transaction allocation decision matrix is summarized in Table 4-1.

Table 4-1. COM+ transaction allocation decision matrix

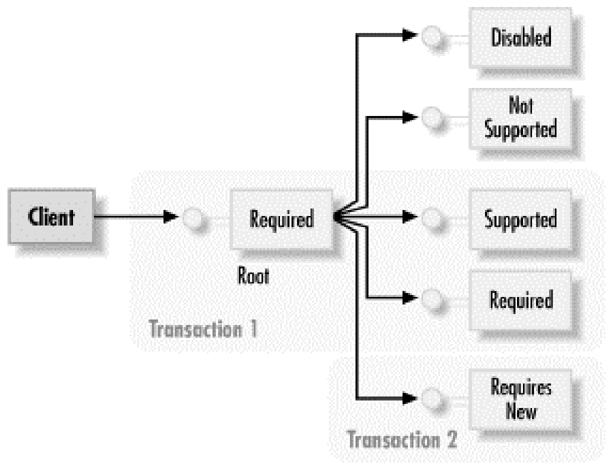
Object transactional support	Creator is in transaction	The object will take part in:
Disabled/Not Supported	No	No Transaction
Supported	No	No Transaction
Required	No	New Transaction (will be the root)
Required New	No	New Transaction (will be the root)
Disabled/Not Supported	Yes	No Transaction
Supported	Yes	Creator's Transaction

Required	Yes	Creator's Transaction
Required New	Yes	New Transaction (will be the root)

Once COM+ determines what transaction to place the object in, that placement is fixed for the life of the object, until the object is released by the client. If the object is not part of a transaction, it will never be part of one. If the object is part of a transaction, it will always be part of that transaction.

Figure 4-8 shows an example of how objects are allocated to transactions. A client that does not have a transaction creates an object configured to require a transaction. COM+ creates a new transaction for that object (Transaction 1), making it the root of the transaction. The object then creates five more objects, each with a different transaction configuration. The objects configured as Disabled and Not Supported are placed outside Transaction 1. The objects market Supported and Required are placed in Transaction 1. However, the object configured as Requires New cannot share its creator's transaction, so COM+ creates a new transaction (Transaction 2) for that object.

Figure 4-8. Allocating objects to transactions based on their configuration and the transaction requirements of the creating object



4.5.1 Transaction Disabled

When you configure a component with transaction support set to Disabled, the component never takes part in any transaction. COM+ also does not consider transactional configuration when deciding on activation context for this component or other components it creates. As a result, the object may or may not share its creator's context, depending on the configuration of other services.

You should be careful when mixing transactional objects with nontransactional objects, as it can jeopardize isolation and consistency. The nontransactional objects may have errors, but because they are not part of the transaction, they cannot affect

transaction outcome (threatens consistency). In addition, the nontransactional objects can act based on information not yet committed (threatens isolation).
The Disabled transaction support setting is useful in two situations. The first situation is when you have no need for transactions. The second is when you want to provide custom behavior and you need to perform your own programmatic transaction support or enlist resources manually. Note that you are not allowed to vote on the outcome of any COM+ transaction; you have to manage your transaction yourself.

4.5.2 Transaction Not Supported

When you configure a component with transaction support set to Not Supported, even though it never takes part in any transaction, COM+ takes into account transactional configuration when deciding on the activation context for this component or other components it creates. As a result, the object shares its creator's context only if the creating object is also configured with Not Supported.
Not Supported is the default value when importing a classic COM component to COM+. Transaction support set to Not Supported is useful when the operations performed by the component are nice to have, but should not abort the transaction that created them if the operations fail. For example, in the ATM use case, printing the receipt is not a critical operation. The withdrawal transaction should commit and the customer should get the bills even if the ATM was unable to print a receipt. In all other circumstances, transactions configured as Not Supported can jeopardize isolation and consistency when mixed with transactional components, for the same reasons discussed when transaction support is set to Disabled.

4.5.3 Transaction Supported

When you configure a component with transaction support set to Supported, the object joins that transaction if the object's creating client has a transaction. If the creating object does not have a transaction, the object does not take part in any transaction. Surprisingly, this awkward setting can be useful. Imagine a situation when you want to propagate a transaction from the creating client of your object to downstream objects your object creates, but your object has no use for transactions itself. If the downstream objects require transaction support and you configure your object to not require a transaction, then the downstream objects will be placed in separate transactions. Setting the transaction support to Supported allows you to propagate the transaction downstream. In all other cases, you should avoid this setting; it can jeopardize consistency and isolation when the

creating client does not have a transaction, but the downstream objects you create still require transaction support and are placed in transactions separate from your client.
Even though the component may not have a direct need for transaction support, it still has to abide by transactional component design guidelines (discussed later in this chapter), which may be a liability if it does not require a transaction. Use this setting judiciously.

4.5.4 Transaction Required

When you configure a component with transaction support set to Required, you state to COM+ that your component requires a transaction to operation properly and that you have no objection to sharing your creator's transaction. If the creating client has a transaction, the object joins it. If the client does not have one, COM+ must create a new transaction for the object, making it the root of the new transaction.
Note that your component's code should operate identically when it is the root and when it just takes part in a transaction. There is no way your object can tell the difference anyway.
Setting transaction support to Required is by far the most commonly used transaction support setting for transactional components. Of course, the component must adhere to the design requirements of a transactional component.

4.5.5 Transaction Requires New

When you configure a component with transaction support set to Requires New, an instance of your component is always the root of a new transaction, regardless of whether its creating client has a transaction or not. This setting is useful when you want to perform transactional work outside the scope of the creating transaction. Examples would be when you want to perform logging or audit operation or when you want to publish events to subscribers, regardless of whether your creating transaction commits or aborts.
You should be extremely careful when using the Requires New setting. Verify that the two transactions (the creating transaction and the one created for your object) do not jeopardize consistency if one aborts and the other commits.
You can also use Requires New when you want your object to control the duration of the transaction because once that object is released, the transaction ends.

4.5.6 Transaction Support IDL Extension

When you import a COM component into the COM Explorer, COM+ selects Not Supported as the default configuration for your component's transaction support. However, transaction support is an intrinsic part of your COM+ component design. COM+ components should specify in the IDL file what their required transaction support is, using a dedicated IDL extension. When you import a COM+ component that uses the IDL extension into the Component Services Explorer, COM+ uses the declared transaction support from the component's type library as the initial value. You can override that value later. For example, if you use the `TRANSACTION_REQUIRED` attribute on your CoClass definition:

```
[
   uuid(94072015-7D6B-4811-BDB5-08983088D9C2),
   helpstring("MyComponent Class"),
   TRANSACTION_REQUIRED
]
coclass MyComponent
{
   [default] interface IMyInterface;
};
```

COM+ selects the Required setting for the component when it is imported to the Component Services Explorer. The following attributes are also available:

- `TRANSACTION_NOT_SUPPORTED`
- `TRANSACTION_SUPPORTED`
- `TRANSACTION_REQUIRES_NEW`

Note that there is no `TRANSACTION_DISABLED` attribute because that attribute is used mostly when importing existing COM components to COM+. To use these IDL extensions you have to include the *mtxattr.h* file in your IDL file.

ATL 7 Transaction Attribute

If you are using attributed ATL 7 project under Visual Studio.NET, you can take advantage of precompiler-specific support for COM+ 1.0 services, using special attributes. If you add a new class to your ATL 7 project and select "ATL COM+ 1.0 Component" from the Add Class dialog, the wizard will let you configure transaction support for your class. Once you select the transaction support (for example, Required), the attributed class will have a custom attribute as part of its declaration:

```
[coclass,
 //other attributes
 custom(TLBATTR_TRANS_REQUIRED,0)]
class MyComponent : IMyInterface
{
    //class declaration
}
```

Before compiling your code, ATL 7 feeds your sources to a special precompiler that parses the attributes and generates conventional, nonattributed ATL source files, including the IDL file. The "new" sources are then fed to the conventional C++ compiler. In that process, the TLBATTR_TRANS_REQUIRED custom attribute is converted to the IDL TRANSACTION_REQUIRED extension.

4.6 Voting on a Transaction

As mentioned before, a transactional object votes whether to commit or abort the transaction by setting the value of a bit in the context object. That bit is called the *consistency* bit. The name is appropriate. Consistency is the only transaction property under the application's objects control. COM+ can manage atomicity and the resource managers guarantee isolation and durability, but only the objects know whether the changes they make to the system state are consistent or if they encounter errors that merit aborting the transaction.

When COM+ creates a transactional object, it puts it in its own private context and sets the context object consistency bit to `TRUE`. As a result, if the object makes no explicit attempt to set the consistency bit to `FALSE`, the object's votes to commit the transaction.

An object can actually share its context with other objects whose transaction setting is set to Disabled.

The object can set the value of the consistency bit by accessing the context object and getting hold of `IContextState` interface, defined as:

```
enum tagTransactionVote
{
   TxCommit= 0,
   TxAbort = TxCommit + 1
}TransactionVote;

interface IContextState : IUnknown
{
   HRESULT SetDeactivateOnReturn([in] BOOL bDeactivate);
   HRESULT GetDeactivateOnReturn([out]BOOL*
pbDeactivate);
   HRESULT SetMyTransactionVote ([in]TransactionVote
txVote);
   HRESULT GetMyTransactionVote ([out]TransactionVote*
ptxVote);
}
```

IContextState is also discussed in Chapter 3, in the context of deactivating JITA objects. IContextState provides the method SetMyTransactionVote( ) used to set the transaction vote. You can pass SetMyTransactionVote( ) the enum value TxCommit, if you want to commit, or the enum value TxAbort, if you want to abort the transaction.

SetMyTransactionVote( ) returns CONTEXT_E_NOTRANSACTION when called by a nontransactional component.

Your object should vote to abort the transaction when it encounters an error that merits aborting the transaction. If all went well, your object should vote to commit the transaction. Example 4-1 shows a typical voting sequence. The object performs some work (the DoWork( ) method) and, according to its success or failure, votes to commit or abort the transaction. If your component decides to abort the transaction, it should return an error code indicating to its client that it aborted the transaction. The client can then decide to retry the transactional operation or handle the error some other way. This is why the component in Example 4-1 returns CONTEXT_E_ABORTING from the method after aborting the transaction. CONTEXT_E_ABORTING is the standard returned value from a component that aborted a transaction.

Example 4-1. Voting on the transaction's outcome by accessing the IContextState interface and calling SetMyTransactionVote( )

```
STDMETHODIMP CMyComponent::MyMethod(  )
{
   HRESULT hres = S_OK;
   hres = DoWork(  );
   //Vote on the transaction outcome

   IContextState* pContextState = NULL;

::CoGetObjectContext(IID_IContextState,(void**)&pContextS
tate);
   ASSERT(pContextState!= NULL);//Not a configured
component

   if(FAILED(hres))
   {
      hres = pContextState-
>SetMyTransactionVote(TxAbort);
      ASSERT(hres != CONTEXT_E_NOTRANSACTION);//No
transaction support
      hres = CONTEXT_E_ABORTING;
   }
   else
   {
      hres = pContextState-
>SetMyTransactionVote(TxCommit);
      ASSERT(hres != CONTEXT_E_NOTRANSACTION);//No
transaction support

   }
   pContextState->Release(  );
   return hres;
}
```

However, what should the client do if an inner object (not the root) votes to abort the transaction? The root object may not know that an inner component has aborted the transaction (and may still vote to commit and return `S_OK` to the client). If `S_OK` is allowed to return to the client, then the client never knows that its request was aborted. To prevent this situation, the interceptor between the root object and its client detects that the transaction is already doomed if an inner object votes to abort and the root object votes to commit and tries to return `S_OK` to the client; it returns `CONTEXT_E_ABORTED` to the client instead.

> One interesting point regarding transaction termination involves exceptions. Any unhandled exception in any object in the transaction (not just the root) terminates the transaction immediately.

4.7 Transactional Object Life Cycle

If a transaction aborts, the intermediate and potentially inconsistent state of the system should be rolled back to ensure consistency. The system state is the data in the resource managers; it also consists of the state of all the objects that took part in the transaction. An object's state is the data members it holds. If the object participated in an aborted transaction, then that object's state should be purged too. The object is not allowed to maintain that state, since it is the product of activities that were rolled back. The problem is that once a transaction ends, even if the object votes to commit, it does not know whether that transaction will actually commit. The DTC still has to collect all the resource managers' votes, conduct the first phase of the two-phase commit protocol, and verify that all of the resource managers vote to commit the transaction. While this process takes place, the object must not accept any new client calls (as part of a new transaction) because the object would act on a system state that may roll back, which would jeopardize consistency and isolation.
To enforce consistency and isolation, once a transaction ends, regardless of its outcome, COM+ releases all the objects that took

part in it. COM+ does not count on objects' having the discipline or knowledge to do the right thing. Besides, even with good intentions, how would the objects know exactly what part of their state to purge?

However, even though the objects are deactivated and released, COM+ remembers their position in the general layout of the transaction: who the root was, who created whom, pointers between objects, and the context, apartment, and process each object belongs to.

When a new method call from the client comes into an object (usually to the root object) that was deactivated at the end of a transaction, COM+ creates a new transaction for that method call and a new instance of the object. COM+ then forwards the call to the new instance. If the object tries to access other objects in the transaction, COM+ re-creates them as well.

In short, COM+ starts a new transaction with new objects in the same *transaction layout* , also called a *transaction stream*. The transaction itself is a transient, short-lived event; the layout can persist for long periods of time. Only when the client explicitly releases the root will the objects really be gone and the transaction layout destroyed.

4.7.1 State-Aware Objects

Because COM+ destroys any object that took part in a transaction at the end of the transaction, transactional objects have to be *state-aware*, meaning they manage their state actively. A state-aware object is not the same as a stateless object. First, as long as a transaction is in progress, the object is allowed to maintain state in memory. Second, the object is allowed to maintain state between transactions, but the state cannot be stored in memory or in the filesystem. Between transactions, a transactional object should store its state in a resource manager. When a new transaction starts, the newly created object should retrieve its state from the resource manager. Accessing the resource manager causes it to auto-enlist with that transaction. When the transaction ends, the object should store its modified state back in the resource manager.

Now here is why you should go though all this hassle: if the transaction aborts, the resource manager will roll back all the changes made during the transaction—in this case, the changes made to the object state. When a new transaction starts, the object again retrieves its state from the resource manager and has a consistent state. If the transaction commits, then the object has a newly updated consistent state. So the object does have state, as long as the object actively manages it.

The only problem now is determining when the object should store its state in the resource manager. When the object is created and placed in a transaction, it is because some other object (its client) tries to invoke a method call on the object. When the call returns, it can be some time until the next method call. Between the two method invocations, the root object can be released or deactivated, ending the transaction. COM+ releases the object, and the object would be gone without ever storing its state back to the resource manager.

The only solution for the object is to retrieve its state at the beginning of every method call and save it back to the resource manager at the end of the method call. From the object's perspective, it must assume that the scope of every transaction is the scope of one method call on it and that the transaction would end when the method returns. The object must therefore also vote on the transaction's outcome at the end of every method.

Because from the object's perspective every method call represents a new transaction, and because the object must retrieve its state from the resource manager, every method definition must contain some parameters that allow the object to find its state in the resource manager. Because many objects could be of the same type accessing the same resource manager, the object must have some key that identifies its state. That key must be provided by the object's client. Typical object identifiers are account numbers and order numbers. For example, the client creates a new transactional order-processing object, and on every method call the client must provide the order number as a parameter, in addition to other parameters. Between method calls, COM+ destroys and re-creates a new instance to serve the client. The client does not know the difference because the two instances have the same consistent state.

Example 4-2 shows a generic implementation of a method on a transactional object. A transactional object must retrieve its state at the beginning of every method and save its state at the end. The object uses an object identifier provided by the client to get and save its state.

The method signature contains an object identifier parameter used to get the state from a resource manager with the `GetState( )` helper method. The object then performs its work using the `DoWork( )` helper method. Then the object saves its state back to the resource manager using the `SaveState( )` method, specifying its identifier. Finally, the object votes on the transaction outcome based of the success of the `DoWork( )` method.

Example 4-2. Implementing a method on a transactional object

```
STDMETHODIMP CMyComponent::MyMethod(PARAM
objectIdentifier)
{
   HRESULT hres = S_OK;
   GetState(objectIdentifier);
   hres = DoWork( );
   SaveState(objectIdentifier);
//Vote on the transaction outcome
   IContextState* pContextState = NULL;

::CoGetObjectContext(IID_IContextState,(void**)&pContextS
tate);
   ASSERT(pContextState!= NULL);//Not a configured
component

   if(FAILED(hres))
   {
      hres = pContextState-
>SetMyTransactionVote(TxAbort);
      ASSERT(hres != CONTEXT_E_NOTRANSACTION);//No
transaction support
      hres = CONTEXT_E_ABORTING;
   }
   else
   {
      hres = pContextState-
>SetMyTransactionVote(TxCommit);
      ASSERT(hres != CONTEXT_E_NOTRANSACTION);//No
transaction support

   }
   pContextState->Release( );
   return hres;
}
```

Note that not all of the object's state can be saved by value to the resource manager. If the state contains pointers to other COM+ objects, `GetState( )` should create those objects and `SaveState( )` should release them. Similarly, if the state contains such resources as database connection, `GetState( )` should acquire a new connection and `SaveState( )` should release the connection.

4.7.2 Transactions and JITA

If the object goes through the trouble of retrieving its state and saving it on every method call, why wait until the end of the transaction to destroy the object? The transactional object should be able to signal to COM+ that it can be deactivated at the end of the method call, even though the transaction may not be over yet. If the object is deactivated between method calls, COM+ should re-create the object when a new method call from the client comes in. The behavioral requirements for a state-aware transactional object and the requirements of a well-behaved JITA object are the same. As discussed in Chapter 3, a well-behaved JITA object should deactivate itself at method boundaries, as well as retrieve and store its state on every method call. Since COM+ already has an efficient mechanism for controlling object activation and deactivation (JITA), it makes perfect sense to use JITA to manage destroying the transactional object and reconnecting it to the client, as explained in Chapter 3.

Every COM+ transactional component is also a JITA component. When you configure your component to require a transaction (including Supported), COM+ configures the component to require JITA as well. You cannot configure your component to not require JITA because COM+ disables the JITA checkbox.

At the end of a method call, like any other JITA object, your transactional object can call `IContextState::SetDeactivateOnReturn( )` to set the value of the done bit in the context object to `TRUE`, signaling to COM+ to deactivate it, as shown in Example 4-3.

Example 4-3. A transactional object deactivating itself at the end of the method

```
STDMETHODIMP CMyComponent::MyMethod(PARAM
objectIdentifier)
{
   HRESULT hres = S_OK;
   GetState(objectIdentifier);
   hres = DoWork(  );
   SaveState(objectIdentifier);

   IContextState* pContextState = NULL;

::CoGetObjectContext(IID_IContextState,(void**)&pContextS
tate);
   ASSERT(pContextState!= NULL);//Not a configured
component

   if(FAILED(hres))
   {
      hres = pContextState-
>SetMyTransactionVote(TxAbort);
      ASSERT(hres != CONTEXT_E_NOTRANSACTION);//No
transaction support
      hres = CONTEXT_E_ABORTING;
   }
   else
   {
      hres = pContextState-
>SetMyTransactionVote(TxCommit);
      ASSERT(hres != CONTEXT_E_NOTRANSACTION);//No
transaction support
```

```
    }
    hres = pContextState->SetDeactivateOnReturn(TRUE);
    pContextState->Release( );
    return hres;
}
```

The done bit is set to `FALSE` by default. If you never set it to `TRUE`, your object is destroyed only at the end of the transaction or when its client releases it. If the object is the root of a transaction, self-deactivation signals to COM+ the end of the transaction, just as if the client released the root object. Of course, by combining transactions with JITA you gain all the benefits of JITA: improved application scalability, throughput, and reliability.

4.7.3 Collecting Objects' Votes

Using JITA has a side effect on your object's transaction vote. When the object is deactivated, the transaction could end while the object is not around to vote. Thus, the object must vote before deactivating itself. When a method call returns, COM+ checks the value of the done bit. If it is `TRUE`, COM+ checks the value of the consistency bit, the object's vote.

COM+ collects the objects' votes during the transaction. Each transaction has a *doomed flag,* which if set to `TRUE` dooms a transaction to abort. COM+ sets the value of a new transaction's doomed flag to `FALSE`.

When an object is deactivated and its vote was to commit, COM+ does not change the current value of the doomed flag. Only if the vote was to abort will COM+ change the doomed flag to `TRUE`. As a result, once set to `TRUE`, the doomed flag value will never be `FALSE` again, and the transaction is truly doomed.

When the root object is deactivated/released, COM+ starts the two-phase commit protocol only if the doomed flag is set to `FALSE`. Note that COM+ does not waste time at the end of a transaction polling objects for their vote. COM+ already knows their vote via the doomed flag.

4.7.4 The IObjectContext Interface

The context object supports a legacy MTS interface, called `IObjectContext`, defined as:

```
interface IObjectContext : IUnknown
{
    HRESULT CreateInstance([in]GUID* rclsid,[in] GUID*
riid,[out,retval]void** ppv);
    HRESULT SetComplete( );
    HRESULT SetAbort( );
    HRESULT EnableCommit( );
    HRESULT DisableCommit( );
```

```
   BOOL IsInTransaction( );
   BOOL IsSecurityEnabled( );
   HRESULT IsCallerInRole([in]BSTR
bstrRole,[out,retval]BOOL* pfIsInRole);
};
```

`IObjectContext` is worth mentioning only because most of the COM+ documentation and examples still use it instead of the new COM+ interface, `IContextState`.

`IObjectContext` has two methods used to vote on a transaction outcome and to control object deactivation. Calling `SetComplete( )` sets the consistency and done bits to `TRUE`. `SetComplete( )` sets the vote to commit and gets the object deactivated once the method returns. `SetAbort( )` sets the vote to abort the transaction and sets the done bit to `TRUE`, causing the object to deactivate when the method returns. COM+ objects should avoid using `IObjectContext` and should use `IContextState` instead.

`IContextState` is fine-tuned for COM+ because it sets one bit at a time. It also verifies the presence of a transaction—it returns an error if the object is not part of a transaction.

COM+ objects written in VB 6.0 have no way of accessing `IContextState` directly. They have to go through `IObjectContext` first and query it for `IContextState`, as shown in Example 4-4. Objects written in Visual Basic.NET can access `IContextState` directly.

Example 4-4. Querying IObjectContext for IContextState

```
Dim objectContext As ObjectContext
Dim contextState As IContextState

Set objectContext = GetObjectContext

'QueryInterface for IContextState:
Set contextState = objectContext
contextState.SetMyTransactionVote TxCommit
```

4.7.5 Method Auto-Deactivation

As shown in Chapter 3, you can configure any method on a JITA object to automatically deactivate the object when it returns. Configuring the method to use auto-deactivation changes the done bit from its default value of `FALSE` to `TRUE`. Because the default value for the consistency bit is `TRUE`, unless you change the context object bits programmatically, auto-deactivation automatically results in a vote to commit the transaction.

However, COM+ examines the `HRESULT` that the method returns. If the `HRESULT` indicates failure, then the interceptor sets the consistency bit to FALSE, as if you voted to abort. This behavior gives you a new programming model for voting and deactivating your object: if you select auto-deactivation for a method, don't take any effort to set any context object bits. Instead, use the method's returned HRESULT:

- If it is S_OK, it is as though you voted to commit. (S_FALSE would also vote to commit.)
- If it indicates failure, it is as though you voted to abort.

When you use auto-deactivation, the programming model becomes much more elegant and concise, and shown in Example 4-5. With auto-deactivation, the object does not have to explicitly vote on the transaction's outcome or deactivate itself. Compare this with Example 4-3. Both have the same effect, but note how elegant, readable, and concise Example 4-5 is.

Example 4-5. Using method auto-deactivation

```
STDMETHODIMP CMyComponent::MyMethod(PARAM
objectIdentifier)
{
   HRESULT hres = S_OK;
   GetState(objectIdentifier);
   hres = DoWork(  );
   SaveState(objectIdentifier);
   return hres;
}
```

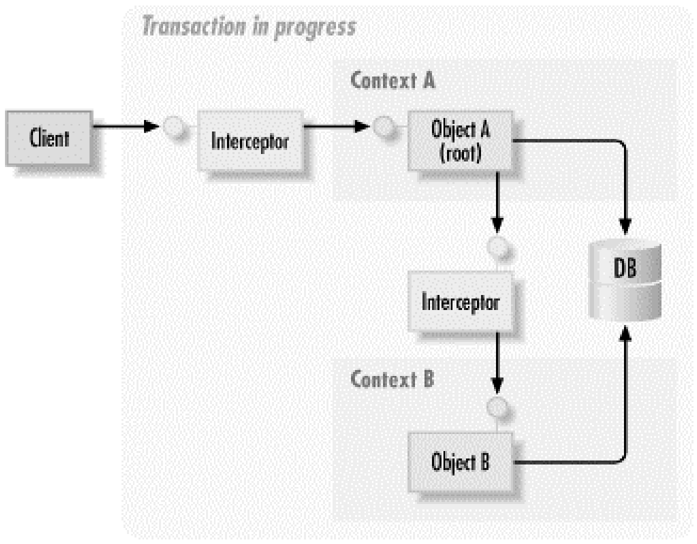
Additionally, the object's client should examine the returned HRESULT. If it indicates failure, then it also indicates that the object voted to abort the transaction; the client should not waste any more time on the transaction because it is doomed.

4.7.6 Object Life Cycle Example

The following simple example demonstrates the important concepts discussed in this section. Suppose a nontransactional client creates Object A, configured with transaction support set to Required. Object A creates Object B, which also requires a transaction. The developers of Object A and Object B wrote the code so that the objects vote and get themselves deactivated on method boundaries. The client calls two methods on Object A and releases it. Object A then releases Object B.

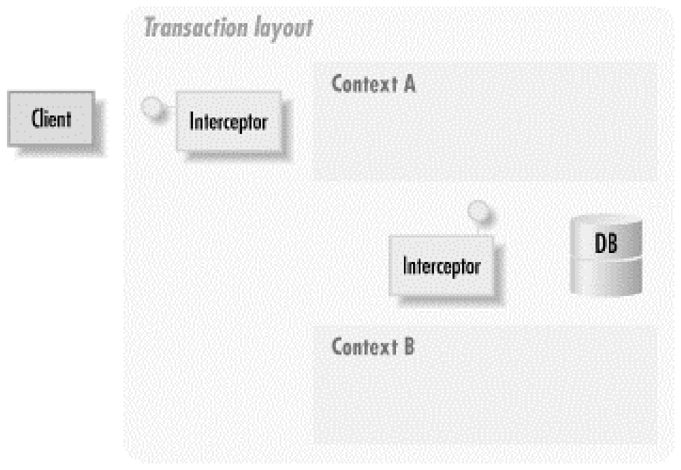
When the client creates Object A, COM+ notes that the client does not have a transaction and that Object A needs transaction support, so COM+ creates a new transaction for it, making Object A the root of that transaction. Object A then goes on to create Object B, and Object B shares Object A's transaction. Note that Object B is in a separate context because transactional objects cannot share a context. Now the transaction layout is established. The transaction layout persists until the client releases Object A, the root of this transaction. Note that both the client and the objects have references to cross-context interceptors, not to actual objects. While a call from the client is in progress, both objects exist (see Figure 4-9) and the transaction layout hosts an actual transaction.

Figure 4-9. Transaction layout while a transaction is in progress



However, between the two method calls from the client, only the transaction layout is maintained; no objects or a transaction are in progress, only interceptors and contexts (see Figure 4-10). When the second call comes in, COM+ creates Object A, and Object A retrieves its state from the resource manager. When Object A accesses Object B to help it process the client request, COM+ creates Object B and hooks it up with the interceptor Object A is using (see Figure 4-9). When the call comes to Object B, it too retrieves its state from the resource manager. When the method returns from Object B, Object B deactivates itself; when the method returns to the client, Object A deactivates itself. When the client releases its reference to Object A, the transaction layout is destroyed, along with the contexts and the interceptors.

Figure 4-10. Transaction layout between method calls



4.8 Designing Transactional Components

Incorporating correct transaction support in your component is an integral part of your design and cannot be done as an afterthought. Supporting transactions is far from simply selecting the correct radio button in the Component Services Explorer. In particular, your object has to be state-aware, actively manage its state, and control its own activation and deactivation, as described in previous sections. You should also design your interfaces to support transactions and to acquire resources in a particular order.

4.8.1 Designing Transactional Interfaces

Interface design is an important factor in designing transactional components. From the object's perspective, method calls demarcate transactions, so you should avoid coupling interface methods to each other. Each method should contain enough parameters for the object to perform its work and decide whether the transaction should commit or abort. In theory, you could build a transactional component that votes on the transaction outcome only after receiving a few method calls. However, in practice, a transaction should not span multiple method calls. You already saw that a transactional object uses JITA and should deactivate itself at method boundaries. COM+ checks the object's vote once it is deactivated. If the interface the object implements requires more than one method call for the object to decide on its vote, then the object could not deactivate itself; it must wait for another call from the client. What should the object do if the transaction suddenly ends (because the root was deactivated or the transaction timed out) and the anticipated call from the client never comes? Waiting for additional information from the client has a serious effect on overall application throughput. While your transaction is in progress, the resource managers involved lock out all other

transactions from the data being modified by your transaction. The other transactions are blocked until your transaction commits or aborts. The longer you wait for client calls that may never come, the more your application's throughput will suffer.
Consider, for example, a poorly designed interface used to handle customer orders:

```
[
 helpstring("Bad design of IOrder interface"),
]
interface IOrder : IUnknown
{
  HRESULT SetOrder([in]DWORD dwOrderNumber);
  HRESULT SetItem([in]DWORD dwItemNumber);
  HRESULT SetCustomerAccount([in]DWORD
dwCustomerAccount);
  HRESULT ProcessOrder(  );
};
```

The interface designer intends for the client to call the `Set( )` methods, supplying the object with the order parameters, and then call `ProcessOrder( )`. The problem with this design is that the transactional object cannot vote on the transaction outcome unless the client calls all the `Set( )` methods and then the `ProcessOrder( )` method, in that sequence. There is no clear delineation of transaction boundaries in this interface design.
The correct way to design the interface while maintaining transaction semantics is:

```
[
 helpstring("Correct design of IOrder interface"),
]
interface IOrder : IUnknown
{
  HRESULT ProcessOrder([in]DWORD dwOrderNumber,[in]DWORD
dwItemNumber,
                       [in]DWORD dwCustomerAccount);
};
```

This interface is also a lot easier to implement. The order number is used to identify the object and allow it to retrieve its corresponding state from the resource manager—in this case, the orders database:

```
STDMETHODIMP COrder::ProcessOrder(DWORD
dwOrderNumber,DWORD wItemNumber,
                                  DWORD
dwCustomerAccount)
{
   HRESULT hres = S_OK;
   GetState(dwOrderNumber);//retrieve the state of the
corresponding
                           //order object
   hres = DoProcessOrder(wItemNumber,dwCustomerAccount);
   SaveState(dwOrderNumber);
   // Using auto-deactivation. No need to vote
explicitly.
   return hres;
}
```

The second interface design yields better performance as well, because there are fewer calls to the object from the client machine, which may be across the network.

4.8.2 Acquiring Resources

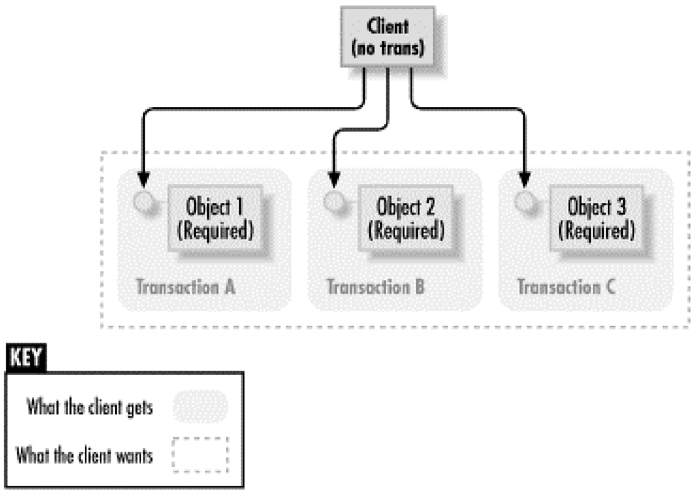
Suppose you have two transactions, T1 and T2, that execute in parallel, and both require access to two resource managers, RM1 and RM2. Suppose T1 has acquired RM1, and T2 has acquired RM2. What would happen if T1 tries to access RM2, and T2 tries to access RM1? You would have a deadlock. Neither transaction is able to proceed. Each needs a resource the other holds to complete its work. Each is blocked and never frees the resource manager it holds.

The solution to this deadly embrace is to be mindful about the order in which objects in your transaction try to acquire resources. You can avoid the deadlock by always trying to acquire the resources in the same order. In the previous example, if both transactions try to acquire RM1 and then RM2, then the first one to actually acquire RM1 will continue on to acquire RM2; the second transaction will be blocked, as it waits for RM1 to be released.

4.9 Nontransactional Clients

Consider the situation in which a nontransactional client creates a few transactional objects, all configured to require transactions. The client would like to scope all its interactions with the objects it creates under one transaction—in essence, to function like the root of that transaction. The problem is that the client is not configured to require transactions (maybe it is a legacy component or maybe it is not even a component, such as a form or a script client), so it cannot have a transaction to include the objects it creates. On the other hand, the objects require transaction support to operate properly, so for every object the client creates, COM+ creates a transaction. As a result, even if the client intended to combine the work of multiple COM+ objects into a single transaction, the net result would be multiple transactions (see Figure 4-11). The real problem now is that each transaction can commit or abort independently. The operations the client performs on the system (using the objects) are no longer atomic, so the client jeopardizes system consistency. Furthermore, even if all objects were under one transaction, how would the client vote to commit or abort that transaction?

Figure 4-11. A nontransactional client ends up with multiple transactions instead of one



There is an elegant and simple solution to this predicament. The solution is to introduce a middleman—a transactional component that creates the objects on behalf of the client. The middleman creates the objects and returns interface pointers back to the client. The middleman objects also should provide the client with ability to commit or abort the transaction.

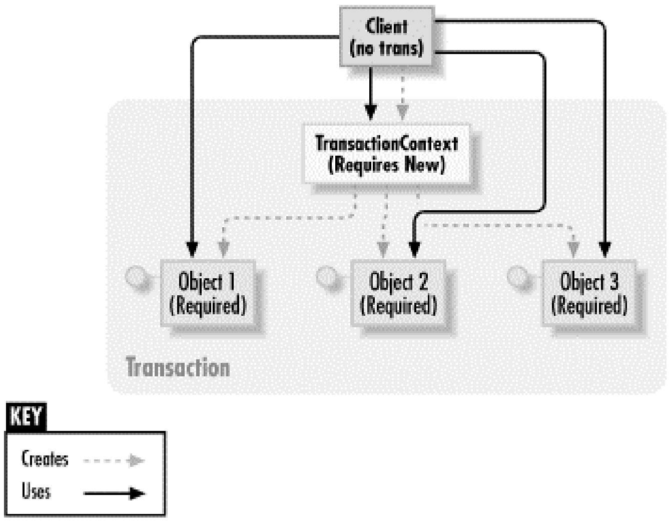
These middleman requirements are generic. Therefore, COM+ provides a readymade middleman called the *transaction context* component. As part of the COM+ Utilities application, COM+ provides two components (one for VB 6.0 and one for C++), each supporting a slightly different interface. A VB 6.0 client should use the `ITransactionContext` interface, creatable via the prog-ID `TxCtx.TransactionContext` (or the class name `TransactionContext`). A C++ client should use the `ITransactionContextEx` interface, which is creatable via the class ID `CLSID_TransactionContextEx`.

The two interfaces are defined as:

```
interface ITransactionContext : IDispatch
{
   HRESULT CreateInstance([in]BSTR pszProgId,[out,
retval]VARIANT* pObject);
   HRESULT Commit( );
   HRESULT Abort( );
};
interface ITransactionContextEx : IUnknown
{
   HRESULT CreateInstance([in]GUID* rclsid,[in]IID* riid,
                          [out,retval]void** pObject);
   HRESULT Commit( );
   HRESULT Abort( );
};
```

These interfaces allow the client to create new component instances and commit or abort the transaction. The two transaction context components are configured to require a new transaction, so they are always the root of that transaction. This configuration also prevents you from misusing the transaction context objects by enlisting them in an existing transaction. All objects created by the transaction context object share the same transaction (see Figure 4-12).

Figure 4-12. Using a middleman, a nontransactional client ends up with one transaction



All the client has to do is create the transaction context object, and then use it to create the other objects via the `CreateInstance( )` method. If the client wants to commit the transaction, it must explicitly call the `Commit( )` method. Once the client calls the `Commit( )` method, the transaction ends on return from the `Commit( )` method. If one of the internal objects votes to abort the transaction before the client calls `Commit( )`, the client's call to `Commit( )` returns with the error code of `CONTEXT_E_ABORT`, indicating that the transaction was already aborted. The client can chose to start a new transaction or handle the error in some other manner.

If the client does not call `Commit( )`, the transaction is aborted, even if all the participating objects voted to commit. This abortion is intentional, to force the client to voice its opinion on the work done by the objects it created. Only the client knows whether their combined work was consistent and legitimate. Apparently, when the client creates the transaction context object, the transaction context object sets the consistency bit to `FALSE` and never deactivates itself. As a result, the transaction is doomed unless the client calls

Commit( ), which causes the transaction context object to change the bit back to TRUE and deactivate itself, thus ending the transaction.
The client can abort the combined transaction by calling the Abort( ) method. The transaction ends on return from the Abort( ) method. The client is also responsible for releasing the references it has on the internal objects created by the transaction context object. It is a good practice to do so even though these objects are released when the transaction ends.
Example 4-6 shows how to use the transaction context object. In the example, the client creates the transaction context object and then uses it to create three transactional objects (as in Figure 4-12). The client votes to commit or abort the transaction, based on the combined success of the method invocations on the three objects.

Example 4-6. Using the transaction context object to create three transactional objects

```
HRESULT hres1 = S_OK;
HRESULT hres2 = S_OK;
HRESULT hres3 = S_OK;

IMyInterface* pObj1= NULL;
IMyInterface* pObj2= NULL;
IMyInterface* pObj3= NULL;
ITransactionContextEx* pTransContext = NULL;

::CoCreateInstance(CLSID_TransactionContextEx,NULL,CLSCTX
_ALL,

IID_ITransactionContextEx,(void**)&pTransContext);

pTransContext-
>CreateInstance(CLSID_MyComponent,IID_IMyInterface,(void*
*)&pObj1);
pTransContext-
>CreateInstance(CLSID_MyComponent,IID_IMyInterface,(void*
*)&pObj2);
pTransContext-
>CreateInstance(CLSID_MyComponent,IID_IMyInterface,(void*
*)&pObj3);

hres1 = pObj1->MyMethod(  );
hres2 = pObj2->MyMethod(  );
hres3 = pObj3->MyMethod(  );

if(S_OK == hres1 && S_OK == hres2 && S_OK == hres3)
   pTransContext->Commit(  );
else
```

```
    pTransContext->Abort( );

pObj1->Release( );
pObj2->Release( );
pObj3->Release( );
pTransContext->Release( );
```

4.10 Transactions and Object Pooling

As discussed in Chapter 3, to speed up performance, your pooled object acquires expensive resources, such as database connections, at creation time and holds onto them while pooled. The problem is that one of the requirements for resource managers is auto-enlistment in transactions. When an object creates a resource such as a database connection, the connection (actually the resource manager) auto-enlists with the object's transaction. A pooled object only creates the resources once, and then the object is called out of the pool to serve clients. Every time the object is retrieved from the pool, it could potentially be part of a different transaction. If the pooled object is forced to re-create the expensive resources it holds to allow them to auto-enlist, that would negate the whole point of using object pooling.

Unfortunately, the only way to combine transactions with a pooled object that holds references to resource managers is to give up on auto-enlistment. The pooled object has to manually enlist the resources it holds in the transactions it participates with.

The pooled object must follow these steps:

1. The object must implement the `IObjectControl` interface. The object needs to manually enlist the resource managers it holds when it is placed in an activation context in its implementation of `IObjectControl::Activate( )`. The object also needs to perform operations in `IObjectControl::Deactivate( )` and `IObjectControl::CanBePooled( )`, explained later.
2. After creating the connection to the resource manager, the pooled object turns off the resource manager's auto-enlistment. This step requires programming against the resource manager API. All resource managers support this functionality, although in slightly different ways and syntax.
3. When the object is called out of the pool to serve a client and is placed in a COM+ context, it must detect whether a transaction is in progress. This detection is done either by calling `IObjectContextInfo::IsInTransaction( )` or calling `IObjectContextInfo::GetTransactionId( )`. If the context the object is placed in is not part of a transaction, the

returned transaction ID is `GUID_NULL`. If a transaction is in progress, the object must manually enlist any resource manager it holds. Enlisting manually is done in a resource-specific manner. For example, in ODBC, the object should call `SQLSetConnectAttr( )` with the `SQL_COPT_SS_ENLIST_IN_DTC` attribute.

   Note that `IObjectControl::Activate( )` is called before the actual call from the client is allowed to access the object. The client call is executed against an object with enlisted resource managers.

4. The object must reflect the current state of its resources and indicate in `IObjectControl::CanBePooled( )` when it can't be reused (if a connection is bad). Returning `FALSE` from `CanBePooled( )` dooms a transaction.

Clearly, mixing resource managers with pooled objects is not for the faint of heart. Besides laborious programming, manually enlisting all resources the object holds every time the object is called from the pool implies a needless performance penalty if the object is called to serve a client in the same transaction as the previous activation. COM+ is aware of the performance penalty and it provides a simple solution. As you saw in Chapter 3, COM+ maintains a pool per component type. However, if a component is configured to use object pooling and require a transaction, COM+ maintains transaction-specific pools for objects of that type.

COM+ actually optimizes object pooling: when the client requesting an object has a transaction associated with it, COM+ scans the pool for an available object that is already associated with that transaction. If an object with the right transaction affinity is found, it is returned to the client. Otherwise, an object from the general pool is returned. In essence, this situation is equivalent to maintaining special subpools containing objects with affinity for a particular transaction in progress. Once the transaction ends, the objects from that transaction's pool are returned to the general pool with no transaction affinity, ready to serve any client.

With this feature, a transactional-pooled object can relieve the performance penalty. Before manually enlisting its resources in a transaction, it should first check to see whether it has already enlisted them in that transaction. If so, there is no need to enlist them again. Your object can achieve that by keeping track of the last transaction ID and comparing it to the current transaction ID using `IObjectContextInfo::GetTransactionId( )`.

Example 4-7 shows a pooled object that takes advantage of COM+ subpooling. In the object's implementation of `IObjectControl::Activate( )`, it gets the current transaction ID.

The object verifies that a transaction is in progress (the transaction ID is not GUID_NULL) and that the current transaction ID is different from the transaction ID saved during the previous activation. If this transaction is new, then the object enlists a resource manager it holds manually. To manually enlist the resource, the object passes the current transaction object (in the form of ITransaction*) to the private helper method EnlistResource( ).
The object saves the current transaction ID in its implementation of IObjectControl::Deactivate( ). The object uses the private helper method IsResourceOK( ) in IObjectControl::CanBePooled( ) to verify that it returns to the pool only if the resource manager is in a consistent state.

Example 4-7. A transactional pooled object manually enlisting a resource manager it holds between activations

```
HRESULT CMyPooledObj::Activate(  )
{
   HRESULT hres = S_OK;
   GUID guidCurrentTras = GUID_NULL;
   hres =  ::CoGetObjectContext(IID_IObjectContextInfo,

(void**)&m_pObjectContextInfo);

   hres = m_pObjectContextInfo-
>GetTransactionId(&guidCurrentTras);
   if(guidCurrentTras!= GUID_NULL && guidCurrentTras !=
m_guidLastTrans)
   {
      ITransaction* pTransaction = NULL;
      hres = m_pObjectContextInfo-
>GetTransaction(&pTransaction);
      hres = EnlistResource(pTransaction);//Helper Method
   }
   return hres;
}
void CMyPooledObj::Deactivate(  )
{
   //Save the current transaction ID
   m_pObjectContextInfo-
>GetTransactionId(&m_guidLastTrans);
   //if no transaction, m_guidLastTrans will be GUID_NULL
   m_pObjectContextInfo->Release(  );
}
BOOL CMyPooledObj::CanBePooled(  )
{
   return IsResourceOK(  );//Helper Method
}
```

Note that though the object is a transactional object, it maintains state across transactions and activations. This maintenance is possible because the object is not really destroyed (only returned to the pool) and its internal state does not jeopardize system consistency.
COM+ does subpooling regardless of whether your transactional-pooled object manages its own resource managers. If your transactional-pooled object does not manually enlist resource managers, then you can just ignore the subpooling.

4.11 Compensating Transactions

Some business operations have a logical undo. Consider the way banks handle bad checks. When you deposit a bad check at the ATM, the bank adds the amount of the check to your account. When the bank discovers the check is bad, it undoes the deposit by deducting an identical amount from your account and returns the check to you in the mail. This logical undo is called a *compensating transaction.* Not every transaction has a compensating transaction, but if it does, you should use caution when incorporating compensating transactions into your application. It is very tricky to use compensating transactions without jeopardizing system consistency. For example, imagine that after depositing the check, you apply for a loan. The bank's loan consultant checks your balance and decides to grant you the loan based on the new increased balance. Once the bank executes the compensating transaction, the system is in an inconsistent state—the account balance is correct, but a loan program is in progress—one that should not have been launched based on the corrected balance. The bank could, of course, perform a compensating transaction for the loan application, except that in the meantime you might have used that loan to start a new business, and so on. As you can see, once the cat is out of the bag, it is difficult to compensate in a comprehensive and consistent manner.
If compensating transactions are bad, why bother with them at all? Compensating transactions are necessary because they enable you to deal efficiently with transactions whose normal execution time is unacceptable. Even though the bad check may bounce after two days, the bank does not expect a customer to wait at the ATM for two days until the check is cleared. Additionally, it is unrealistic to keep a lock on the customer's account for two days because no other operation on the account can take place until the depositing transaction is done. The bank has to take the chance and use a compensating transaction as a safety net. The bank, in this case, trades transaction throughput for a small, calculated risk in system consistency.

In general, compensating transactions are useful when the transaction for which they compensate is potentially long. Compensating transactions offer a high throughput alternative, allowing you to maintain locks in the resource managers for a minimum amount of time.

4.12 Transaction Execution Time

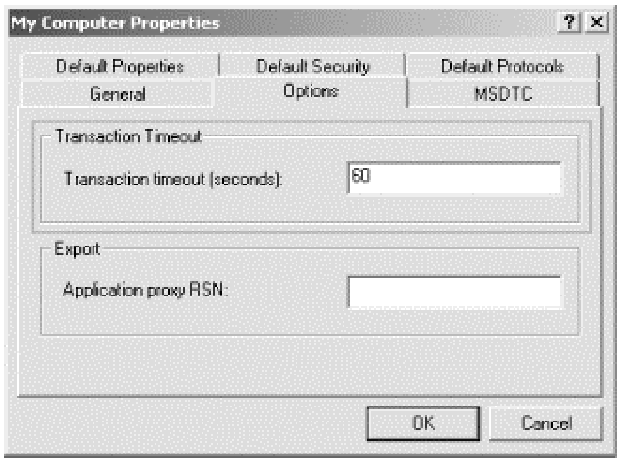
Transaction execution time should be minimal. The reason is obvious: a transaction occupies expensive resources. As long as the transaction executes, no other transaction can access those resources. Every resource manager the transaction accesses has to lock relevant data, isolating that transaction from the rest of the world. As long as the locks are held, nobody else can access the data. The more transactions per second your application can process, the better its scalability and throughput.

Transaction execution usually requires, at most, a few seconds. For lengthy operations, consider using a short transaction backed up by a compensating transaction.

COM+ allows you to configure a maximum execution time for your transactions. If your transaction reaches that timeout, COM+ aborts it automatically. Transaction timeouts prevent resource manager deadlocks from hanging the system. Eventually, one of the two transactions deadlocking each other would reach the timeout and abort, allowing the other transaction to proceed.

You can configure two kinds of transaction timeouts. The first is a machine-wide parameter called the *global transaction timeout* . The global timeout applies to all transactions on that machine. You configure the global timeout by right-clicking on the My Computer icon in the Component Services Explorer, selecting Properties from the context menu, and selecting the Options tab (see Figure 4-13). The default timeout is set to 60 seconds, but you can set it to any value you like, up to 999 seconds. A global timeout set to zero means an infinite timeout. Transactions on that machine never time out. Infinite timeout is useful mostly for debugging, when you want to try to isolate a problem in your business logic by stepping through your code and you do not want the transaction you debug to time out while you figure out the problem. Be extremely careful with infinite timeout in all other cases because it means there are no safeguards against transaction deadlocks.

Figure 4-13. Setting global transaction timeout



Transaction Timeout

Transaction timeout (seconds): 60

Export

Application proxy RSN:

You can also configure transaction timeout at the component level, on its Transactions tab. Component-level transaction timeout is disabled by default, and you have to explicitly enable it. Component-level transaction timeout means that any transaction this component is part of must end within the time specified, or else COM+ aborts it. Obviously, the component-level timeout is effective only if it is less than the global timeout. The default component-level timeout is set by COM+ to zero, which indicates infinity. You can use component-level timeout in two cases. The first case is during development, when you want to test the way your application handles aborted transactions. By setting the component-level timeout to a small value (such as one second), you cause your transaction to fail and can thus observe your error handling code. The second case in which you set the component-level transaction timeout to be less than the global timeout is when you believe that the component is involved in more than its fair share of resource contention, resulting in deadlocks. In that case, you should abort the transaction as soon as possible and not wait for the global timeout to expire.

4.13 Tracing Transactions

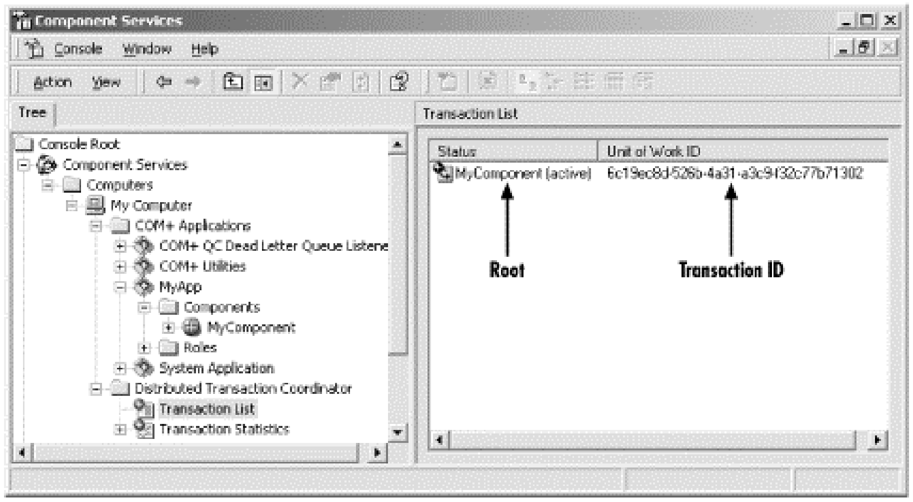
Sometimes, during development, or perhaps during deployment for logging purposes, you may want to trace the current transaction ID under which your object executes. COM+ provides you with two ways to retrieve the transaction ID, programmatically and administratively, using the Component Services Explorer.
To trace the current transaction ID programmatically, you should use `IObjectContextInfo::GetTransactionId( )`. Example 4-8 shows how to trace the current transaction ID to the output window in the debugger.

Example 4-8. Tracing the current transaction ID to the output window

```
HRESULT hres = S_OK;
GUID guidTransactionID = GUID_NULL;
IObjectContextInfo* pObjectContextInfo = NULL;

hres = ::CoGetObjectContext(IID_IObjectContextInfo,
                            (void**)&pObjectContextInfo);

ASSERT(pObjectContextInfo != NULL); //a non-configure
object maybe?


hres = pObjectContextInfo-
>GetTransactionId(&guidTransactionID);


pObjectContextInfo->Release(  );

if(guidTransactionID == GUID_NULL)
{
   ATLTRACE("The object does not take part in a
transaction");
}
else
{
   USES_CONVERSION;
   WCHAR pwsGUID[150];
   ::StringFromGUID2(guidTransactionID,pwsGUID,150);
   ATLTRACE("The object takes place in transaction with
ID %s ",W2A(pwsGUID));
}
```

As long as a transaction is in progress, you can view its transaction ID in the Component Services Explorer when using the administrative method. Under the My Computer icon in the Component Services Explorer is the Distributed Transaction Coordinator (DTC) folder. Expand the DTC folder and select the Transaction List item. The right pane in the Component Services Explorer contains a list of all the transactions executing on your machine (see Figure 4-14).

Figure 4-14. The transaction list view



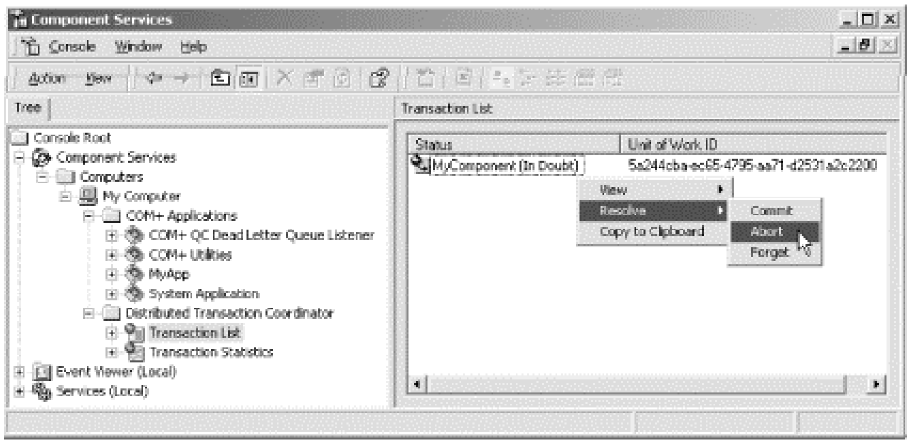
Status	Unit of Work ID
MyComponent (active)	6c19ec8d-528b-4a31-a3c9f32c77b71302

The Component Services Explorer presents a few bits of information on every transaction. The Status column contains the type of the root component and the status of the transaction, and the Unit of Work ID column contains the transaction ID.

4.14 In-Doubt Transactions

Sometimes COM+ (actually, the DTC) is unable to decide on the fate of a transaction. This indecisiveness is usually the result of some unexpected catastrophe. One possible catastrophe is network failure after the root object is deactivated, but before the DTC could conduct the two-phase commit protocol with remote resource managers. Another possible catastrophe is when a resource manager's machine crashes in the middle of the two-phase commit protocol. In those cases, the transaction is said to be *in-doubt.* COM+ cannot decide on the fate of in-doubt transactions. It is up to the system administrator to manually resolve those transactions, using the Component Services Explorer. COM+ lists the in-doubt transactions under the DTC folder, on the Transaction List pane. An in-doubt transaction has the comment (In Doubt) in its status column. The system administrator should right-click on the in-doubt transaction and select Resolve from the pop-up menu. COM+ offers three options to resolve a transaction: Commit, Abort, or Forget (see Figure 4-15).

Figure 4-15. Resolving in-doubt transactions



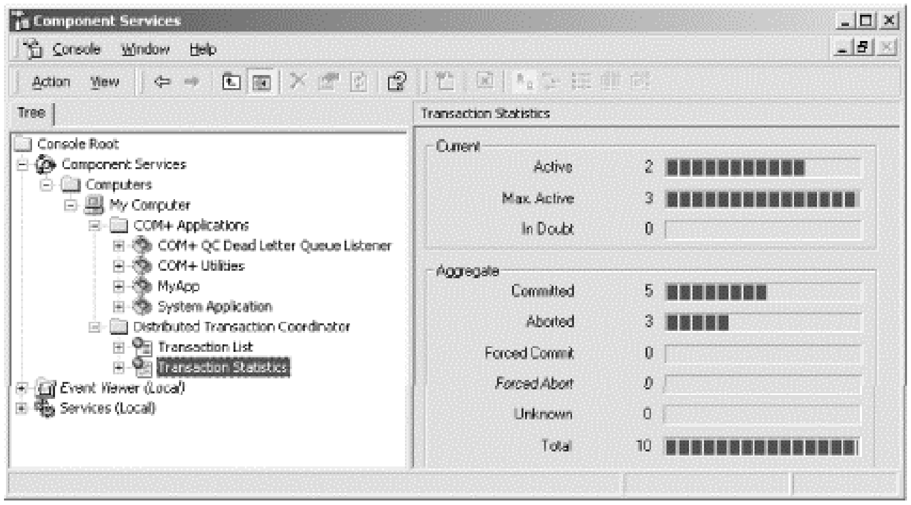
Status	Unit of Work ID
MyComponent (In Doubt)	5a244cba-ec65-4795-aa71-d2531a2c2200

When the administrator selects Commit or Abort, COM+ instructs all accessible resource managers that took part in the transaction to commit or abort, respectively. Later on, when the rest of the resource managers become available, your system administrator should use an administrative utility to launch a compensating transaction on those resources.
The interesting resolving option is Forget. By choosing to forget about the transaction, your administrator instructs COM+ to do absolutely nothing with this transaction besides remove it from the list. The administrator is willing to accept the inconsistent state the system is in, and does not wish to commit or abort the transaction. Forgetting a transaction may be useful in some esoteric scenarios. Imagine that while a transaction was in doubt, some administrator manually changed entries in the database because he did not wish to wait for the transaction to be resolved. In such a case, your application administrator may choose to forget about the original transaction and accept the current state.

4.15 Transaction Statistics

The Component Services Explorer can show you various transactions statistics. You view the statistics by selecting the Transaction Statistics item in the DTC folder (see Figure 4-16). The statistics view contains various numbers regarding the currently executing transactions, as well as aggregated numbers resulting from all transactions that took place since the last machine reboot.

Figure 4-16. The Transaction Statistics item



Current

Active	2
Max. Active	3
In Doubt	0

Aggregate

Committed	5
Aborted	3
Forced Commit	0
Forced Abort	0
Unknown	0
Total	10

The following list contains explanations of the various statistics:

Active

The total number of currently executing transactions.

Max. Active

The maximum number of transactions that were active concurrently since the last reboot. This number can be used as a crude throughput indicator.

In Doubt

The total number of transactions currently in doubt.

Committed

The total number of transactions committed since the last reboot.

Aborted

The total number of transactions aborted since the last reboot.

Forced Commit

The total number of transaction that were in doubt that the administrator resolved by forcing to commit. A value other then zero is usually the result of a catastrophe that was resolved manually.

Forced Abort

The total number of transactions that were in doubt that the administrator resolved by forcing to abort. A value other then zero is usually the result of a catastrophe that was resolved manually.

Unknown

The total number of transactions whose fate is unknown.

Total

The total number of transactions created since the last reboot.

The statistics are useful when you try to calibrate various application parameters, such as pool sizes, to maximize throughput. An important throughput indicator is the number of transactions processed in a given amount of time. You can get that number and quality metrics, such as the number of aborted transactions, from the statistics view.

4.16 COM+ Transactions Pitfalls

I'll end this chapter by pointing out a few more pitfalls you should be aware of when designing and developing transactional components in COM+. Some of these pitfalls have already been implied elsewhere in this chapter, but elaborating on a pitfall is always a good idea.

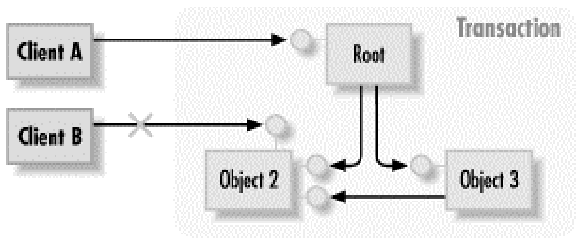
4.16.1 Accessing Nontransactional Resources

A transactional component should avoid accessing resources that are not resource managers. Typical examples are the filesystem, the Registry, network calls, and user interaction such as printouts or message boxes. The reason is obvious—if the transaction aborts, changes made to those transaction-ignorant resources will persist and jeopardize system consistency.

4.16.2 Passing Subroot Objects to Clients

You should avoid passing subroot objects to any client outside your transaction, be it the client that created the root or any other client. You have to avoid this by design because COM+ allows you to stumble into the pitfall. The problem with sharing subroot objects with clients outside of your transaction is that at any moment the client that created the root object can release the root object. A COM+ transaction requires a root to function, and the root designation does not change, no matter how the transaction is started. With the root gone, the transaction layout is defective. In Figure 4-17, any call from Client B to Object 2 will fail with the error code `CONTEXT_E_OLDREF`. The only thing Client B can do is release its reference to Object 2.

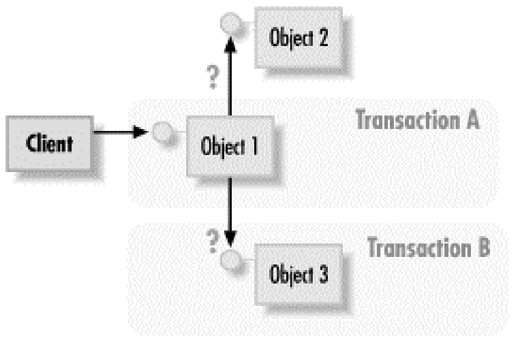
Figure 4-17. Avoid passing or sharing subroot objects with any client outside the transaction



4.16.3 Accessing Objects Outside the Transaction

You should avoid accessing COM+ objects outside your transaction, whether those objects are part of another transaction or not. Look at the objects layout in Figure 4-18.

Figure 4-18. Accessing COM+ objects outside your transaction can jeopardize system consistency



In this figure, Object 1 has access to Object 2 and Object 3, both outside its transaction. The problem is that Transaction A could abort and Transaction B could commit. Object 3 acts based on its interaction with an object from an aborted transaction, and therefore Object 3 jeopardizes system consistency when its transaction commits. Similarly, when Object 1 accesses Object 2 (which does not have a transaction at all), Object 2 may operate based on inconsistent state if Transaction A aborts. In addition, the interaction between Object 1 and Object 2 is not well defined. For example, should Object 1 abort its transaction if Object 2 returns an error? For these reasons, objects should only access other objects within the same transaction.

Chapter 5. COM+ Concurrency Model

Employing multiple threads of execution in your application opens the way for many benefits impossible to achieve using just a single thread. These benefits include:

Responsive user interface

Your application can process user requests (such as printing or connecting to a remote machine) on a different thread than that of the user interface. If it were done on the same thread, the user interface would appear to hang until the other requests were processed. Because the user interface is on a different thread, it can continue to respond to the user's request.

Enhanced performance

If the machine your application runs on has multiple CPUs and the application is required to perform multiple calculation-intensive independent operations, the only way to use the extra processing power is to execute the operations on different threads.

Increased throughput

If your application is required to process incoming client requests as fast at it can, you often spin off a number of worker threads to handle requests in parallel.

Asynchronous method calls

Instead of blocking the client while the object processes the client request, the object can delegate the work to another thread and return control to the client immediately.

In general, whenever you have two or more operations that can take place in parallel and are different in nature, using multithreading can bring significant gains to your application.

The problem is that introducing multithreading to your application opens up a can of worms. You have to worry about threads deadlocking themselves while contesting for the same resources, synchronize access to objects by concurrent multiple threads, and be prepared to handle object method re-entrancy. Multithreading bugs and defects are notoriously hard to detect, reproduce, and eliminate. They often involve rare race conditions (in which multiple threads write and read shared data without appropriate access synchronization), and fixing one problem often introduces another. Writing robust, high performance multithreading object-oriented code is no trivial matter. It requires a great deal of skill and discipline on behalf of the developers.

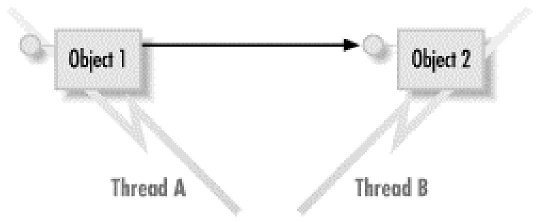
Clearly there is a need to provide some concurrency management service to your components so you can focus on the business problem at hand, instead of on multithreading synchronization issues. The classic COM concurrency management model addresses

the problems of developing multithreaded object-oriented applications. However, the classic COM solution has its own set of deficiencies.
COM+ concurrency management service addresses the problems with the classic COM solution. It also provides you with administrative support for the service via the Component Services Explorer.
This chapter first briefly examines the way classic COM solves concurrency and synchronization problems in classic object-oriented programming, and then introduces the COM+ concurrency management model, showing how it improves classic COM concurrency management. The chapter ends by describing a new Windows 2000 threading model, the neutral threaded apartment, and how it relates to COM+ components.

5.1 Object-Oriented Programming and Multiple Threads

The classic COM threading model was designed to address the set of problems inherent with objects executing in different threads. Consider, for example, the situation depicted in Figure 5-1. Under classic object-oriented programming, two objects on different threads that want to interact with each other have to worry about synchronization and concurrency.

Figure 5-1. Objects executing on two different threads



Object 1 resides in Thread A and Object 2 resides in Thread B. Suppose that Object 1 wants to invoke a method of Object 2, and that method, for whatever reason, must run in the context of Thread B. The problem is that, even if Object 1 has a pointer to Object 2, it is useless. If Object 1 uses such a pointer to invoke the call, the method executes in the context of Thread A.
This behavior is the direct result of the implementation language used to code the objects. Programming languages such as C++ are completely thread-oblivious—there is nothing in the language itself to denote a specific execution context, such as a thread. If you have a pointer to an object and you invoke a method of that object, the compiler places the method's parameters and return address on the calling thread's stack—in this case, Thread A's stack. That does not have the intended effect of executing the call in the context of

Thread B. With a direct call, knowledge that the method should have executed on another thread remains in the design document, on the whiteboard, or in the mind of the programmer.

The classic object-oriented programming (OOP) solution is to post or send a message to Thread B. Thread B would process the message, invoke the method on Object 2, and signal Thread A when it finished. Meanwhile, Object 1 would have had to block itself and wait for a signal or event from Object 2 signifying that the method has completed execution.

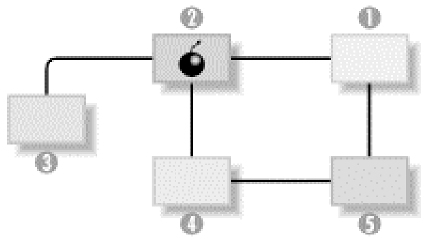
This solution has several disadvantages: you have to handcraft the mechanism, the likelihood of mistakes (resulting in a deadlock) is high, and you are forced to do it over and over again every time you have objects on multiple threads.

The more acute problem is that the OOP solution introduces tight coupling between the two objects and the synchronization mechanism. The code in the two objects has to be aware of their execution contexts, of the way to post messages between objects, of how to signal events, and so on. One of the core principals of OOP, encapsulation or information hiding, is violated; as a result, maintenance of classic multithreaded object-oriented programs is hard, expensive, and error-prone.

That is not all. When developers started developing components (packaging objects in binary units, such as DLLs), a classic problem in distributed computing raised its head. The idea behind component-oriented development is building systems out of well-encapsulated *binary* entities, which you can plug or unplug at will like Lego bricks. With component-oriented development, you gain modularity, extensibility, maintainability, and reusability. Developers and system designers wanted to get away from monolithic object-oriented applications to a collection of interacting binary components. Figure 5-2 shows a product that consists of components.

The application is constructed from a set of components that interact with one another. Each component was implemented by an independent vendor or team. However, what should be done about the synchronization requirements of the components? What happens if Components 3 and 1 try to access Component 2 at the same time? Could Component 2 handle it? Will it crash? Will Component 1 or Component 3 be blocked? What effect would that have on Component 4 or 5? Because Component 2 was developed as a standalone component, its developer could not possibly know what the specific runtime environment for the components would be. With that lack of knowledge, many questions arise. Should the component be defensive and protect itself from multiple threads accessing it? How can it participate in an application-wide synchronization mechanism that may be in place? Perhaps Component 2 will never be accessed simultaneously by two threads in this application; however, Component 2's developer cannot know this in advance, so it may choose to always protect the component, taking an unnecessary performance hit in many cases for the sake of avoiding deadlocks.

Figure 5-2. Objects packaged in binary units have no way of knowing about the synchronization needs of other objects in other units



5.2 Apartments: The Classic COM Solution

The solution used by classic COM is deceptively simple: each component declares its synchronization needs. Classic COM makes sure that instances (objects) of that class always reside in an execution context that fits their declared requirements, hence the term *apartment*. A component declares its synchronization needs by assigning a value to its `ThreadingModel` named-value in the Registry. The value of `ThreadingModel` determines the component's threading model. The available values under classic COM are `Apartment`, `Free`, `Both` or no value at all.

Components that set their threading model to `Apartment` or leave it blank indicate to COM that they cannot handle concurrent access. COM places these objects in a single-threaded environment called a *single-threaded apartment* (STA). STA objects always execute on the same STA thread, and therefore do not have to worry about concurrent access from multiple threads.

Components that are capable of handling concurrent access from multiple clients on multiple threads set their threading model to `Free`. COM places such objects in a *multithreaded apartment* (MTA).

Components that would like to always be in the same apartment as their client set their threading model to `Both`. Note that a `Both` component must be capable of handling concurrent access from multiple clients on multiple threads because its client may be in the MTA.

As discussed in Chapter 2, classic COM marshals away the thread differences between the client and an object by placing a proxy and stub pair in between. The proxy and stub pair blocks the calling thread, performs a context switch, builds the calling stack on the

object's thread, and calls the method. When the call is finished, control returns to the calling thread that was blocked.
Although apartments solve the problem of methods executing outside their threads, they contribute to other problems, specifically:

- Classic COM achieves synchronization by having an object-to-thread affinity. If an object always executes on the same thread, then all access to it is synchronized. But what if the object does not care about thread affinity, but only requires synchronization? That is, as long as no more than one thread accesses the object at a given time, the object does not care which thread accesses it.
- The STA model introduces a situation called *object starvation.* If one object in a STA hogs the thread (that is, performs lengthy processing in a method call) then all other objects in the same STA cannot serve their clients because they must execute on the same thread.
- Sharing the same STA thread is an overkill of protection—calls to all objects in a STA are serialized; not only can clients not access the same object concurrently, but they can't access different objects in the same thread concurrently.
- Even if a developer goes through the trouble of making its object thread-safe (and marks it as using the `Free` threading model), if the object's client is in another apartment, the object still must be accessed via a proxy-stub and incur a performance penalty.
- Similarly, all access to an object marked as `Both` that is loaded in a STA is serialized for no reason.
- If your application contains a client and an object each in different apartments, you pay for thread context-switch overhead. If the calling pattern is frequent calls to methods with short execution times, it could kill your application's performance.
- MTA objects have the potential of deadlock. Each call into the MTA comes in on a different thread. MTA objects usually lock themselves for access while they are serving a call. If two MTA objects serve a call and try to access each other, a deadlock occurs.
- Local servers that host MTA objects face esoteric race conditions when the process is shut down while they are handling new activation requests.

5.3 Activities: The COM+ Innovation

The task for COM+ was not only to solve the classic OOP problems but also to address the classic COM concurrency model deficiencies and maintain backward compatibility. Imagine a client calling a method on a component. The component can be in the same context as the client, in another apartment or a process on the same machine, or in a process on another machine. The called component may in turn call other components, and so on, creating a string of nested calls. Even though you cannot point to a single thread that carries out the calls, the components involved do share a *logical* thread of execution.

Despite the fact that the logical thread can span multiple threads, processes, and machines, there is only one root client. There is also only one thread at a time executing in the logical thread, but not necessarily the same physical thread at all times.

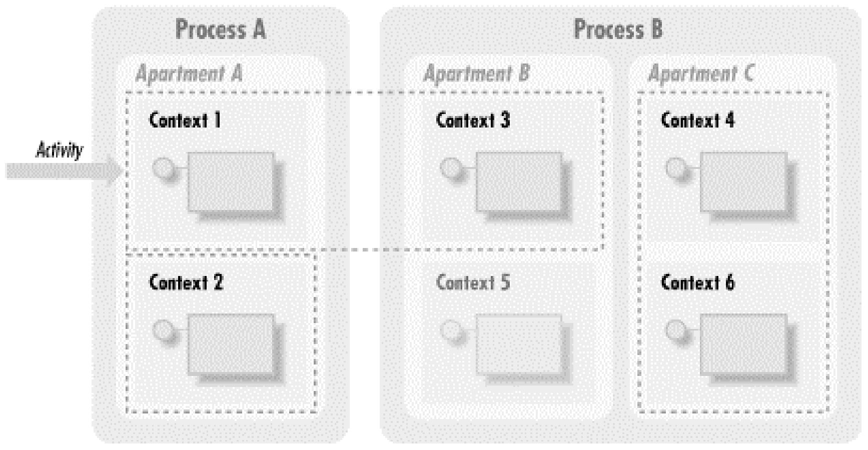
The idea behind the COM+ concurrency model is simple, but powerful: instead of achieving synchronization through physical thread affinity, COM+ achieves synchronization through logical thread affinity. Because in a logical thread there is just one physical thread executing in any given point in time, logical thread affinity implies physical threads synchronization as well. If a component is guaranteed not to be accessed by multiple logical threads at the same time, then synchronization to that component is guaranteed. Note that there is no need to guarantee that a component is always accessed by the same logical thread. All COM+ provides is a guarantee that the component is not accessed by more than one logical thread at a time.

A logical thread is also called a *causality,* a name that emphasizes the fact that all of the nested calls triggered by the root client share the same cause—the root client's request on the topmost object. Due to the fact that most of the COM+ documentation refers to a logical thread as causality, the rest of this chapter uses causality too. COM+ tags each causality with its own unique ID—a GUID called the *causality ID*.

To prevent concurrent access to an object by multiple causalities, COM+ must associate the object with some sort of a lock, called a *causality lock*. However, should COM+ assign a causality lock per object? Doing so may be a waste of resources and processing time, if by design the components are all meant to participate in the same activity on behalf of a client. As a result, it is up to the component developer to decide how the object is associated with causality-based locks: whether the object needs a lock at all, whether it can share a lock with other objects, or whether it requires a new lock. COM+ groups together components than can share a causality-based lock. This grouping is called an *activity*.

It is important to understand that an activity is only a logical term and is independent of process, apartment, and context: objects from different contexts, apartments, or processes can all share the same activity (see Figure 5-3).

Figure 5-3. Activities (indicated by dashed lines) are independent of contexts, apartments, and processes



Within an activity, concurrent calls from multiple causalities are not allowed and COM+ enforces this requirement. Activities are very useful for MTA objects and for neutral threaded apartment (NTA) objects, a new threading model discussed at the end of the chapter; these objects may require synchronization, but not physical thread affinity with all its limitations. STA objects are synchronized by virtue of thread affinity and do not benefit from activities.

5.3.1 Causality-Based Lock

To achieve causality-based synchronization for objects that take part in an activity, COM+ maintains a causality-based lock for each activity. The activity lock can be owned by at most one causality at a time. The activity lock keeps track of the causality that currently owns it by tracking that causality's ID. The causality ID is used as an identifying key to access the lock. When a causality enters an activity, it must try to acquire the activity lock first by presenting the lock with its ID. If the lock is already owned by a different causality (it will have a different ID), the lock blocks the new causality that tries to enter the activity. If the lock is free (no causality owns it or the lock has no causality ID associated with it), the new causality will own it. If the causality already owns that lock, it will not be blocked, which allows for callbacks. The lock has no timeout associated with it; as a result, a call from outside the activity is blocked until the current causality exits the activity. In the case of more than one causality trying to enter the activity, COM+ places all pending causalities in a queue and lets them enter in the activity in order.

The activity lock is effective process-wide only. When an activity flows from Process 1 to Process 2, COM+ allocates a new lock in Process 2 for that activity, so that attempts to access the local objects in Process 2 will not have to pay for expensive cross-process or cross-machine lookups.

An interesting observation is that a causality-based lock is unlike any other Win32 API-provided locks. Normal locks (critical sections, mutexes, and semaphores) are all based on a physical thread ID. A normal physical thread-based lock records the physical thread ID that owns it, blocking any other physical thread that tries to access it, all based on physical thread IDs. The causality-based lock lets all the physical threads that take part in the same logical thread (same causality) go through; it only blocks threads that call from different causalities. There is no documented API for the causality lock.
Activity-based synchronization solves the classic COM deadlock of cyclic calling—if Object 1 calls Object 2, which then calls Object 3, which then calls Object 1, the call back to Object 1 would go through because it shares the same causality, even if all the objects execute on different threads.

5.3.2 Activities and Contexts

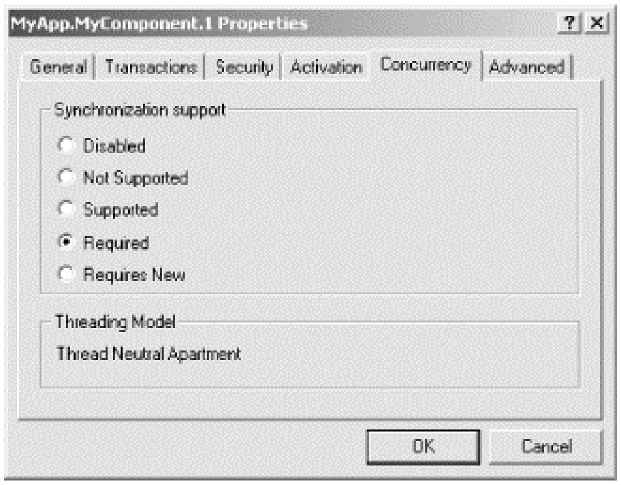
So how does COM+ know which activity a given object belongs to? What propagates the activity across contexts, apartments, and processes? Like almost everything else in COM+, the proxy and stub pair does the trick.
COM+ maintains an identifying GUID called the *activity ID* for every activity. When a client creates a COM+ object that wants to take part in an activity and the client has no activity associated with it, COM+ generates an activity ID and stores it as a property of the context object (discussed in Chapter 2). A COM+ context belongs to at most one activity at any given time, and maybe none at all.
The object that created the activity ID is called the *root* of the activity. When the root object creates another object in a different context—say Object 2—the proxy to Object 2 grabs the activity ID from the context object and passes it to the stub of Object 2, potentially across processes and machines. If Object 2 requires synchronization, its context uses the activity ID of the root.

5.4 COM+ Configuration Settings

Every COM+ component has a tab called Concurrency on its properties page that lets you set the component synchronization requirements (see Figure 5-4). The possible values are:

- Disabled
- Not Supported
- Supported
- Required
- Requires New

Figure 5-4. The Concurrency tab lets you configure your component's synchronization requirements



Synchronization support

- ( ) Disabled
- ( ) Not Supported
- ( ) Supported
- (●) Required
- ( ) Requires New

Threading Model

Thread Neutral Apartment

The synchronization is activity based, as explained before. These settings are used to decide in which activity the object will reside in relation to its creator. As you may suspect, the way the synchronization values operate is completely analogous to the transaction support configuration values, discussed in Chapter 4. An object can reside in any of these activities:

- In its creator's activity: the object shares a lock with its creator.
- In a new activity: the object has its own lock and starts a new causality.
- In no activity at all: there is no lock, so concurrent access is allowed.

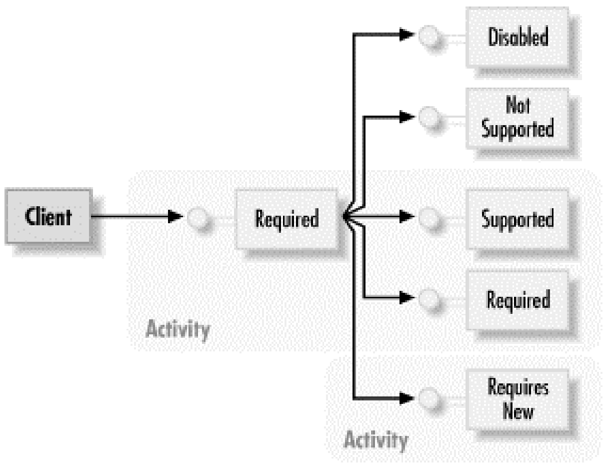
An object's activity is determined at creation time, based on the activity of the creator and the configured requirement of the object. For example, if the object is configured to have a synchronization setting of Required, it will share its creator's activity if it has one. If the creator does not have an activity, then COM+ creates a new activity for the object. The effects of this synchronization support are defined in Table 5-1.

Table 5-1. Determinants of an object's activity

Object synchronization support	Is creator in activity?	The object will take part in:
Disabled/Not Supported	No	No Activity
Supported	No	No Activity
Required	No	New Activity
Required New	No	New Activity
Disabled/Not Supported	Yes	No Activity
Supported	Yes	Creator's Activity
Required	Yes	Creator's Activity
Required New	Yes	New Activity

Figure 5-5shows an example of activity flow. In the figure, a client that does not take part in an activity creates an object configured with Synchronization = Required. Since the object requires an activity and its creator has none, COM+ makes it the root of a new activity. The root then goes on to create five more objects. Two of them, configured with Synchronization = Required and Synchronization = Supported, are placed in the same activity as the root. The two components configured with Synchronization = Not Supported and Synchronization = Disabled will have no activity. The last component is configured with Synchronization = Requires New, so COM+ creates a new activity for it, making it the root of its own activity.

Figure 5-5. Allocating objects to activities based on their configuration and the activity of their creator



You may be asking yourself why COM+ bases the decision on the object's activity partly on the object's creating client. The heuristic technique COM+ uses is that the calling patterns, interactions, and synchronization needs between objects usually closely match their creation relationship.
An activity lasts as long as the participating objects exist, and its lifetime is independent of the causalities that enter and leave it. A causality is a transient entity that lasts only as long as the client's call is in progress. The activity to causality relationship is analogous to the transaction layout to transaction relationship described in Chapter 4.

5.4.1 Synchronization Disabled

When you choose to disable synchronization support, you are instructing COM+ to ignore the synchronization requirements of the component in determining context for the object. As a result, the object may or may not share its creator's context.

You can use the Disabled setting when migrating a classic COM component to COM+. If that component was built to operate in a multithreaded environment, it already has a synchronization mechanism of some sort, and you must disable the synchronization attribute to maintain the old behavior.
In addition, if you disable synchronization on a component, that component should never access a resource manager because it might require the activity ID for its own internal locking.

5.4.2 Synchronization Not Supported

An object set to Not Supported never participates in an activity, regardless of causality. The object must provide its own synchronization mechanism. This setting is only available for components that are nontransactional and do not use JITA. I recommend avoiding this setting because it offers nothing to the developer except restrictions.

5.4.3 Synchronization Supported

An object set to Supported will share its creator's activity if it has one, and will have no synchronization of its own if the creator does not have one.
This is the least useful setting of them all because the object must provide its own synchronization mechanism in case its creator does not have an activity. You must make sure that the mechanism does not interfere with COM+ activities when COM+ provides synchronization. As a result, it is more difficult to develop the component.

5.4.4 Synchronization Required

When an object is set to Required, all calls to the object will be synchronized, and the only question is whether your object will have its own activity or share its creator's activity. When COM+ creates the object, it looks at the activity status of its creator. If the creator has an activity, COM+ extends the creator's activity boundary to include the new object. Otherwise, COM+ creates a new activity. If you don't care about having your own activity, always use this setting.

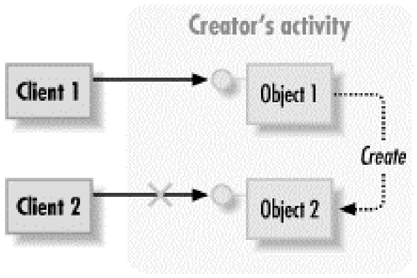
5.4.5 Synchronization Requires New

When an object is set to Requires New, the object must have a new activity, distinct from the creator's activity, and have its own lock. The object will never share its context with its creator. In fact, this is one of the sure ways of ensuring that your object will always be created in its own context.

5.4.6 Required Versus Requires New

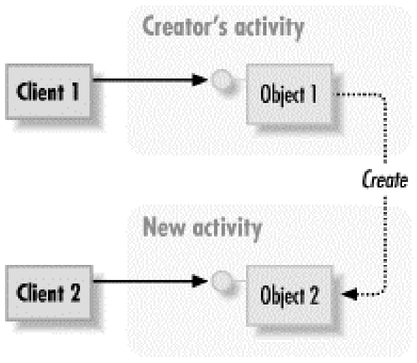
Deciding that your object requires synchronization is usually straightforward. If you anticipate multiple clients on multiple threads trying to access your object and you don't want to write your own synchronization mechanism, you need synchronization. The more difficult question to answer is whether your object should require its own activity lock or whether you should configure it to use the lock of its creator. Try basing your decision on the calling patterns to your object. Consider the calling pattern in Figure 5-6. Object 2 is configured with synchronization set to Required and is placed in the same activity as its creator, Object 1. In this example, besides creating Object 2, Object 1 and Object 2 do not interact with each other.

Figure 5-6. Sharing activities enable calls to be accepted from another client



While Client 1 accesses Object 1, Client 2 comes along, wanting to call methods on Object 2. Because Client 2 has a different causality, it will be blocked. In fact, it could have safely accessed Object 2, since it does not violate the synchronization requirement for the creating object, Object 1.
On the other hand, if you were to configure Object 2 to require its own activity by setting the Synchronization to Requires New, the object could process calls from other clients at the same time as Object 1 (see Figure 5-7).

Figure 5-7. In this calling pattern, having a separate activity for the created object enables it to service its clients more efficiently



However, calls from the creator object (Object 1) to Object 2 will now potentially block and will be more expensive because they must cross context boundaries and pay the overhead of trying to acquire the lock.

5.5 Activities and JITA

Components that use JITA are required to be accessed by one client at a time. If two clients could call a JITA component simultaneously, one would be left stranded when the object was deactivated at the time the first method call returned. COM+ enforces synchronization on components that use JITA. The Concurrency tab for components that have JITA enabled will only allow you to set your component to Required or Requires New. In other words, the component must share the activity of its creator or require a new activity. The other options are disabled on the Concurrency tab. Once you disable JITA, you can set synchronization to other values.

5.6 Activities and Transactions

Transactional objects also allow access to them by only one client at a time. Synchronization is required to prevent the case in which one client on one thread tries to commit a transaction while another client on a second thread tries to abort it. As a result, every transaction should have a synchronization lock associated with it. On the other hand, having more than one lock in a given transaction is undesirable—spinning off a new activity for an object that is added to an existing transaction means always paying for the overhead for checking the activity lock before accessing the object. That check is redundant because no two causalities are allowed in the same transaction anyway. In fact, when an object requires a new transaction, it could still reuse the same causality lock of its creator and allow the activity to flow into the new transaction. COM+ therefore enforces the fact that a given transaction can only be part of one activity (note that an activity can still host multiple transactions).
In addition, as discussed in Chapter 4, transactional objects always use JITA (COM+ automatically enables JITA for a transactional object). The use of JITA is only optional for nontransactional objects. Table 5-2 summarizes the synchronization values as a product of the transaction and JITA setting. Note that the only case when a transactional component can start a new activity is when that component is also configured to be the root of a new transaction.

Table 5-2. Component's available synchronization settings

Transaction setting	JITA setting	Available synchronization setting
Disabled	Off	All
Not Supported	Off	All
Disabled	On	Required or Requires New
Not Supported	On	Required or Requires New
Supported	On	Required
Required	On	Required
Requires New	On	Required or Requires New

5.7 Tracing Activities

COM+ makes it easy for an object to retrieve its activity identity, using the context object interface `IObjectContextInfo`, with the method:

```
HRESULT GetActivityID(GUID* pguidActivityID);
```

If the object does not take part in an activity, the method returns `GUID_NULL`. Retrieving the activity ID is useful for debugging and tracing purposes.

Example 5-1 demonstrates activity ID tracing.

Example 5-1. Tracing the activity ID

```
HRESULT hres = S_OK;
GUID guidActivityID = GUID_NULL;
IObjectContextInfo* pObjectContextInfo = NULL;

hres = ::CoGetObjectContext(IID_IObjectContextInfo,
                            (void**)&pObjectContextInfo);

ASSERT(pObjectContextInfo != NULL);//a non-configure
object maybe?

hres = pObjectContextInfo-
>GetActivityId(&guidActivityID);

pObjectContextInfo->Release( );

if(guidActivityID == GUID_NULL)
{
   TRACE("The object does not take part in an activity");
}
else
{
   USES_CONVERSION;
   WCHAR pwsGUID[150];

   ::StringFromGUID2(guidActivityID,pwsGUID,150);
   TRACE("The object takes place in activity with ID
%s",W2A(pwsGUID));
}
```
COM+ provides the activity ID via another interface, called `IObjectContextActivity`, obtained by calling `CoGetObjectContext( )`.
`IObjectContextActivity` has just one method, `GetActivityId( )`, used exactly like the method of the same name in the example.

5.8 The Neutral Threaded Apartment

The neutral threaded apartment (NTA) is a new threading model available only on Windows 2000. Although it is not specific to COM+ (classic COM objects can also take advantage of the NTA), the NTA is the recommended threading model for most COM+ objects that do not have a user interface.
The NTA has evolved to address a deficiency in the classic COM MTA threading model: suppose you have an STA client accessing an MTA object. Under classic COM, all cross-apartment calls have to be marshaled via a proxy/stub pair. Even though the object could have handled the call on the client STA thread, the call is marshaled. The stub performed an expensive thread context switch to an RPC thread to access the MTA objects.
There was clearly a need for an apartment that every thread in the process could enter without paying a heavy performance penalty. This is what the NTA is: an apartment that every COM-aware thread can enter. In every process, there is exactly one NTA. The NTA is subdivided (like any other apartment) into contexts. COM objects that reside in the NTA set their threading model value in the Registry to `Neutral`.
Much like an MTA object, an object marked as neutral will reside in the NTA, regardless of its creator's apartment. Calls into the NTA are marshaled, but only light-weight proxies are used (to do cross COM+ context marshaling, if needed) because no thread-context switch is involved. A method call on an NTA object is executed on the caller's thread, be it STA or MTA based.
No thread calls the NTA home, and the NTA contains no threads, only objects. Threads can't call `CoInitializeEx( )` with a flag saying NTA, and no such flag exists. When you create a thread, you still must assign it to an STA of its own or to the MTA.

5.8.1 The NTA and Other COM Threading Models

When you mark your object as `Neutral`, it will always reside in the NTA, regardless of the location of its creating client. When you mark your object as `Both`, if the object's creator is an NTA object, the object will reside in the NTA as well. If your NTA object creates other objects marked as `Apartment`, the location of the creating thread may affects where those objects reside. Table 5-3 presents the potential results when NTA clients create other objects. It also shows the resulting object apartment, based on the object threading model and the thread the NTA client runs on. You can also see from Table 5-3 that components marked as `Neutral` will always be in the NTA, regardless of the apartment of their creator.

Table 5-3. Apartment activation policy

Object is\Client is:	Apartment	Free	Both	Neutral	Not specified
STA, not main	Current STA	MTA	Current STA	NTA	Main STA
Main STA	Main STA	MTA	Main STA	NTA	Main STA
MTA	Host STA	MTA	MTA	NTA	Main STA
Neutral (on STA thread)	On that STA thread	MTA	NTA	NTA	Main STA
Neutral (on MTA thread)	Host STA	MTA	NTA	NTA	Main STA

The NTA model obeys the COM rule specifying that all objects must be marshaled outside the apartment/context boundary, just like any other apartment. If you have to manually marshal an object outside the NTA, use the Global Interface Table (the GIT) or the GIT wrapper class, presented in Chapter 2.

Finally, the NTA offers improved DCOM performance because incoming calls from remote machines to NTA objects can execute directly on the thread that handles the incoming remote call, without a thread context switch.

5.8.2 COM+ and Threading Model

Your COM+ component should run in the STA if any one of the following statements is valid:

- Your COM+ component displays a user interface or it relies on having a message loop pump messages to it. Your component relies on the STA thread message pump.
- Your COM+ component uses Thread Local Storage (TLS), a thread-specific heap allocated off the thread stack. It must run in the STA because TLS relies on having the thread affinity the STA provides.
- Your component was provided by a third party as a COM component and marked as `Apartment`. You want to import it to your COM+ application so that it shares your application settings, such as security and process, and is part of your application's MSI file. You should not change the threading model, because you do not know how much thread affinity the component requires.
- Your component is developed using Visual Basic 6.0.

Your COM+ component should use the `Both` threading model if the creating client is in the STA or MTA, but not the NTA; it makes very frequent method calls; and the calls have short duration. By using `Both`, you will avoid cross-apartment marshaling, an overhead that may hinder performance under this scenario.
In all other cases, your COM+ component should use the `Neutral` threading model. You will need to use activity-based synchronization to provide synchronization to your component.
You should avoid using the `Free` threading model for your component because running in the NTA will offer the same throughput without the additional thread context switch involved with calls into the MTA. Only legacy components imported into COM+ should use `Free` as the threading model.

5.9 Summary

Activity-based synchronization is a simple and elegant concurrency management service that provides both an administrative support and a straightforward programming model. For most cases, if your design calls for using multithreading, configure your component to require synchronization, and COM+ will do the rest. That way, you can devote your development effort to the business problem (instead of the synchronization issues), and the resulting code is robust. COM+ synchronization is almost a formal way of eliminating potentially hard-to-solve synchronization defects.
The first five chapters present the basic COM+ component services: application activation, instance management, transaction support, and concurrency management. The rest of the chapters describe higher-level COM+ services (security, queued components, and loosely coupled events). I call these services "high level" because they all rely and interact with the basic services. Before you learn these high-level services, you need to be familiar with programmatic configuration of COM+ services, the subject of the next chapter.

Chapter 6. Programming the COM+ Catalog

COM+ stores the information about your applications, your components' configuration and physical locations, global machine settings, and every other bit of data COM+ requires to operate in a repository called the *COM+ Catalog.*
The Catalog exposes COM+ interfaces and components that allow you to access the information it stores. Anything you can do visually with the Component Services Explorer, you can do programmatically as well—from exporting COM+ applications to doing fine-grained configuration such as enabling auto-deactivation on a method. In fact, the Component Services Explorer and the various wizards are merely handy user-interface wrappers around the Catalog interfaces and objects.
This chapter covers the COM+ Catalog programming model and provides you with useful code samples you can use as a starting point for automating all tasks of administrating COM+ applications and services.

6.1 Why Program the Catalog?

Some of the more advanced features of COM+ lack support in the Component Services Explorer and are available only by configuring your components programmatically. These features are largely tied in with COM+ Events (discussed in Chapter 9) and include COM+ events filtering and managing transient subscriptions to COM+ events.
Programming the COM+ Catalog gives you access to much more than advanced services. By learning to program the Catalog, you can provide your system administrators with helper utilities that automate tedious tasks. These helpers interact with the underlying Catalog on the administrators' behalf, saving them the trouble of learning how to use the Component Services Explorer and presenting them with familiar terminology from the application domain. A typical example is adding a new user to the system: you can create a utility to programmatically add the user to an appropriate role, without requiring the administrator to launch and interact with the Component Services Explorer (role-based security is discussed in Chapter 7). You can even create a utility to enable your system administrator to remotely deploy, administer, and configure your product's components and applications on different machines (by accessing those machines' Catalogs) while remaining at his desk.

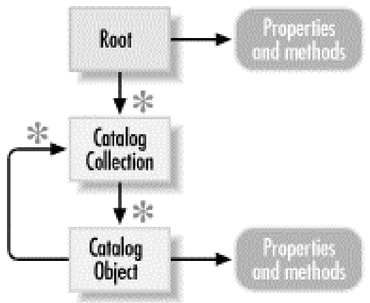
You can also capture user input or deployment-specific information during your application setup and fine-tune your application configuration in the Catalog. The user sees just one installation process because all access to the Catalog can be done programmatically.
Finally, during your component development, you benefit greatly from automating such tasks as starting and shutting down applications. You will see an example of that later in the chapter.

6.2 The Catalog Programming Model

The information stored in the Catalog is structured similarly to its layout in the Component Services Explorer. Data items in the Catalog are more or less where you would expect to find them according to their visual representation. In general, folders in the Component Services Explorer (such as applications, roles, components, and interfaces) correspond to COM+ Catalogcollections. A *catalog collection* is a collection of items of some uniform kind. Every collection has a string identifying it, called the *collection name.* One example of a catalog collection is the `Applications` collection. The items in a collection are called *catalog objects.* You can add or remove catalog objects in a collection, just as you can add or remove items in a Component Services Explorer folder. For example, when you add a catalog object to the `Applications` collection, you are actually adding a COM+ application.
Every catalog object in a collection exposes properties that you can read or configure. The catalog object properties are similar or identical to the properties available on the properties page in the Component Services Explorer for that particular item type. For example, the properties of a catalog object from the `Applications` collections are COM+ application properties—such as activation mode (server or library) or idle time management timeouts.
Essentially, all you ever do with the COM+ Catalog is locate the collection you are interested in, iterate over its catalog objects, find the object you are looking for, modify its properties, and save your changes. In practice, the Catalog's programming model is uniform, whether you iterate over the `Applications` collection or the `Components` collection of a specific application. The Catalog exposes a hierarchy of predefined collections and objects, and you program against those collections and objects. The Catalog interfaces are dual COM interfaces, which enables you to call them from within administration scripts.
Abstracted, the Catalog design pattern is depicted in Figure 6-1. Each catalog collection may contain many catalog objects. A collection's sole purpose is to allow you to iterate over the objects it

contains. A collection has no properties you can configure, much like how a folder in the Component Services Explorer has no properties. You only set the properties of catalog objects. Each catalog object has a set of properties and methods you can invoke. Each catalog object can also give you access to other collections associated with it. For example, in the `Applications` collection, every application object has a `Components` collection associated with it, analogous to the Components folder under every application in the Component Services Explorer. As you can see in Figure 6-1, the Catalog has a root object. The root is special kind of a catalog object, and the Catalog has only one root object. The root object also has properties and methods you can call. The root object gives you access to *top-level collections* such as the `Applications` collection. The root object is your gateway to the COM+ Catalog and is available as a COM object.

Figure 6-1. The COM+ Catalog design pattern



All three object types (collection, object, and root) support three different interfaces. Every catalog collection supports the `ICatalogCollection` interface, and every catalog object supports the `ICatalogObject` interface. The `ICatalogCollection` interface is designed to iterate over a collection of `ICatalogObject` interface pointers. The `ICatalogObject` allows you to access the object's properties by referring to each property by a predetermined name (an identifying string). In addition, each catalog object has a key that you use to get the collections associated with that catalog object.

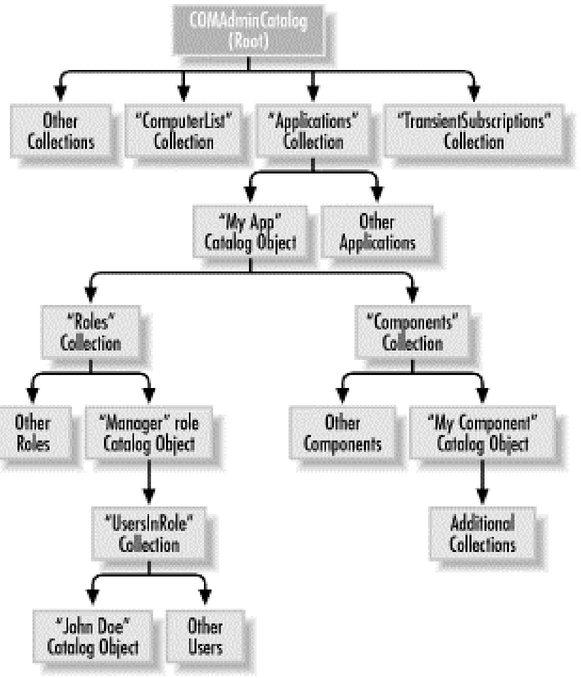
The Catalog root supports a third interface called `ICOMAdminCatalog`, with special root-level methods and properties. The `ICOMAdminCatalog` interface lets you access the top-level collections. When accessing the top-level collections, there is no need for a key because there is only one root object.

The goal of this design pattern is to have an extremely extensible programming model. Because all collections and objects support the same interfaces, regardless of the actual collection or object, they are all accessed and manipulated the same way. If in the future there is a need to define new collections (such as new services in future versions of COM+), the same structure and programming model would be able to define and use the new collections and catalog objects.

6.3 Catalog Structure

This section discusses the Catalog structure and the names of the items in it, not the semantics of these items. Some of these items have already been covered in the previous chapters, and some are covered in subsequent chapters. The COM+ Catalog's actual structure, from the root down to the component level, is mapped out in Figure 6-2. Each collection has a predefined identifying name, whereas catalog objects' names are defined by the user. The root of the Catalog gives you access to top-level collections such as the `Applications` and `TransientSubscription` collections (see Chapter 9). You can also access less useful collections such as the communication protocols used by DCOM or all of the in-proc servers (COM objects in a DLL) installed on the machine. Another top-level collection shown in Figure 6-2 is the `ComputerList` collection—a list of all the computers that the Component Services Explorer is configured to manage.

Figure 6-2. The COM+ Catalog structure, from the root down to the component level

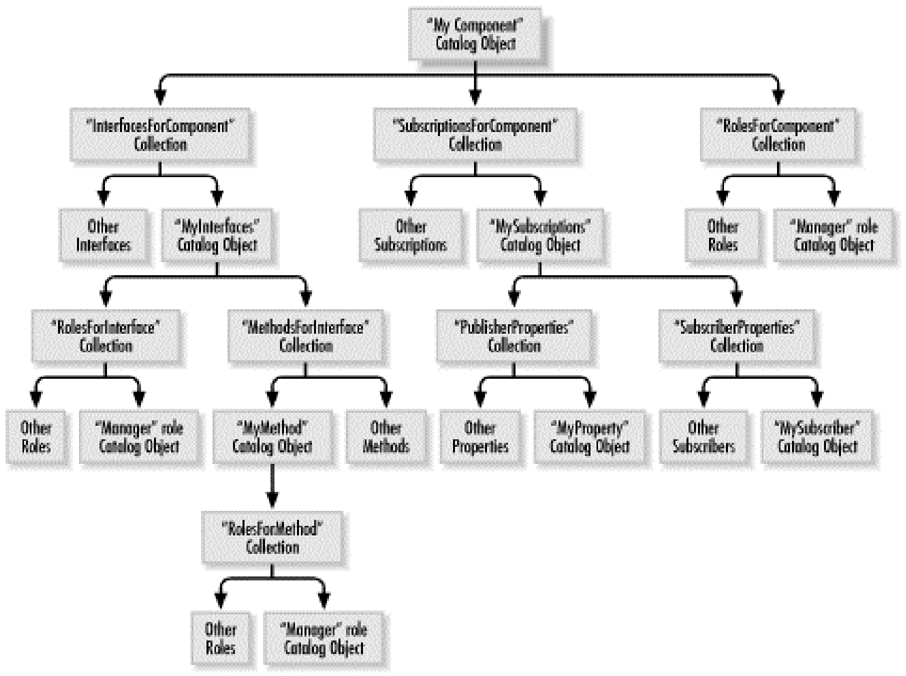


The `Applications` collection, as the name implies, contains all the COM+ applications installed on the machine. A catalog object in the `Applications` collection allows you to set the properties of a particular COM+ application. It also gives you access to two other collections: the Roles and the Components collections. As mentioned previously, every folder in the Component Services Explorer corresponds to a catalog collection. Just as every application in the Component Services Explorer has a *Roles* and *Components* subfolder, a catalog object representing an application can give you access to these two collections.

The Roles collection contains a catalog object for each role defined in the application. Chapter 7 discusses role-based security at length. Every catalog object in the Roles collection lets you set its properties (such as the role name and description) and give you access to a collection of users associated with that role, called the UsersInRole collection. Every catalog object in the UsersInRole collection represents a user that was added to that role. As you can see in Figure 6-2, the objects in the UsersInRole collection do not have any collections associated with them.

The Components collection contains a catalog object for each component in the application. You can programmatically configure all the properties available on the properties page of a component in the Component Services Explorer. Every component catalog object can give you access to three collections: the InterfacesForComponent collection, the SubscriptionForComponent collection, and the RolesForComponent collection (see Figure 6-3).

Figure 6-3. Every component catalog object has an elaborate structure under it


"My Component" Catalog Object
"InterfacesForComponent" Collection
"SubscriptionsForComponent" Collection
"RolesForComponent" Collection
Other Interfaces
"MyInterfaces" Catalog Object
Other Subscriptions
"MySubscriptions" Catalog Object
Other Roles
"Manager" role Catalog Object
"RolesForInterface" Collection
"MethodsForInterface" Collection
"PublisherProperties" Collection
"SubscriberProperties" Collection
Other Roles
"Manager" role Catalog Object
"MyMethod" Catalog Object
Other Methods
Other Properties
"MyProperty" Catalog Object
Other Subscribers
"MySubscriber" Catalog Object
"RolesForMethod" Collection
Other Roles
"Manager" role Catalog Object

The InterfacesForComponent collection contains a catalog object for every interface the component supports. Every interface catalog object gives you access to its properties and to two collections—one is called the `RolesForInterface` collection, used to iterate over the roles that were granted access for this interface, and the second collection is the `MethodsForInterface` collection. The `MethodsForInterface` collection contains a catalog object for each method on that interface. Each method catalog object can give you access to its properties and to the roles associated with that method, in a collection called `RolesForMethod`.

Going back to the collections accessible from every component catalog object, the `RolesForComponent` collection lets you access the roles associated with that component, and the `SubscriptionsForComponent` collection contains a catalog object per a subscription to a COM+ Event (discussed in Chapter 9). Every subscription object is associated with two collections—the `PublisherProperties` and the `SubscriberProperties` collection.

The only role objects that have collections of users associated with them are in the `Roles` collection accessible from every application object (see Figure 6-2). The component, interface, and method level role objects do not have user collections associated with them (see Figure 6-3).

One more bit of COM+ Catalog trivia—every catalog object always has at least three collections associated with it: the `RelatedCollectionInfo`, `PropertyInfo`, and `ErrorInfo` collections. These collections were omitted from Figure 6-2 and Figure 6-3 for the sake of clarity. The `RelatedCollectionInfo` collection is used for advanced iterations over the Catalog, allowing you to write generic recursive iteration code that discovers at runtime which collections a particular catalog object is associated with. The `PropertyInfo` collection is used to retrieve information about the properties that a specified collection supports. The `ErrorInfo` collection can provide extensive error information for dealing with errors in methods that update more than one catalog object at once, so you can find out exactly which object caused the error. This chapter does not discuss these three advanced collections.

When programming against the COM+ Catalog structure, you need not memorize the Catalog intricate structure. You can just follow the intuitive structure of the Component Services Explorer and simply provide the correct collection name, while using Figures 6-2 and 6-3 as reference navigation maps.

6.4 Interacting with the Catalog

Besides understanding the Catalog physical structure, you need to be familiar with how to interact with the three Catalog interfaces and object types (root, collection, and object). This section will walk you through a few programming examples and demonstrate most of what you need to know when programming the Catalog.

6.4.1 The Catalog Root Object

The starting point for everything you do with the Catalog is the root object. You create the root object with the class ID of `CLSID_COMAdminCatalog` (or the prog-ID of `COMAdmin.COMAdminCatalog`) and obtain an interface pointer to the `ICOMAdminCatalog`interface. You use the `ICOMAdminCatalog`interface pointer to either invoke root-level methods or access one of the top-level collections by calling the `GetCollection( )` method, defined as:

```
[id(1)] HRESULT GetCollection([in]BSTR
bstrCollectionName,
                              [out,retval]IDispatch**
ppCatalogCollection);
```

You can use `ICOMAdminCatalog::GetCollection( )` to access only the top-level collections (such as `Applications`) shown in Figure 6-2. Accessing lower level collections is done differently, and you will see how shortly. `GetCollection( )` returns an `ICatalogCollection` pointer to the specified collection. Once you get the collection you want, you can release the root object. Example 6-1 shows how to access the `Applications` collection by creating the root object and calling `ICOMAdminCatalog::GetCollection( )`.

Example 6-1. Accessing a top-level collection such as Applications

```
HRESULT hres = S_OK;
ICOMAdminCatalog* pCatalogRoot  = NULL;
ICatalogCollection* pApplicationCollection = NULL;

hres =
::CoCreateInstance(CLSID_COMAdminCatalog,NULL,CLSCTX_ALL,

IID_ICOMAdminCatalog,(void**)&pCatalogRoot);
hres = pCatalogRoot-
>GetCollection(_bstr_t("Applications"),

(IDispatch**)&pApplicationCollection);
```

```
pCatalogRoot->Release( ); //You don't need the root any
more

/* use pApplicationCollection */
```

Later, you will see other uses for the `ICOMAdminCatalog` interface besides just accessing a top-level collection.

6.4.2 The ICatalogCollection Interface

Every collection in the COM+ Catalog implements the `ICatalogCollection` interface. As mentioned previously, the `ICatalogCollection` interface is used to iterate over a collection of catalog objects. The `ICatalogCollection` interface supports several methods and properties. The main methods it supports are `Populate( )`, `Add( )`, `Remove( )`, `SaveChanges( )`, and `GetCollection( )`. The main properties are `Count` and `Item`.

After obtaining a collection interface (be it a top-level or a lower-level collection), the first thing you need to do is call the `Populate( )` method. The `Populate( )` method reads the information from the Catalog into the collection object you are holding, populating the collection with data for all the items contained in the collection.

If you want to change the collection by adding or removing a catalog object, use the `Add( )` or `Remove( )` methods. The `Add( )` method is defined as:

```
[id(2)] HRESULT Add([in]IDispatch* pCatalogObject);
```

It accepts just one parameter—a pointer to the catalog object you wish to add to the collection.

The `Count` property returns the number of objects in the collection and must be prefixed by a `get_` when accessed from C++ (there are plenty of examples later in the chapter).

The `Item` property is defined as:

```
[id(1),propget] HRESULT Item([in] long lIndex,
                             [out,retval]IDispatch**
ppCatalogObject);
```

This property returns a pointer to a catalog object, given its index. Collection indexes are zero-based, not one-based, meaning the first element has index `zero` and the last has index `count-1`. You can now write a `for` loop that iterates over the entire collection, retrieving one item at a time. Once you have a pointer to a catalog object, you can read and change its named properties.

The `Remove( )` method is defined as:

```
[id(3)] HRESULT Remove(long lIndex);
```

It accepts an index in the collection identifying the object you wish to remove.

Whatever change you make to the collection (adding or removing objects or modifying object properties) will not take effect unless you call the `SaveChanges( )` method. It is a common pitfall to write code that iterates correctly over a collection, modifies it, and releases all the objects properly—but forgets to call SaveChanges( ). Next time your Catalog administration code executes and no apparent change has taken place, go back and make sure you called SaveChanges( ).

Finally, the GetCollection( ) method is defined as:

```
[id(4] HRESULT GetCollection([in] BSTR
bstrCollectionName,
                              [in] VARIANT varObjectKey),
                              [out,retval]IDispatch**
ppCollection);
```

This method is used to retrieve a catalog collection associated with a particular catalog object. As explained previously, a catalog object can have catalog collections associated with it (see Figures 6-2 and 6-3). The catalog object interface has no means for providing those collections; you get them by calling GetCollection( ) on the collection containing the object. GetCollection( ) accepts a key value as a parameter, so that it can identify the object whose collection you wish to access. Note that ICOMAdminCatalog::GetCollection( ) did not require a key because the top-level collections are already named uniquely. In the case of a lower level collection, many objects will have collections associated with them, all named the same. For example, if you iterate over the Applications collection, you will find that each item (a catalog object) is an application and each of them has a Components collection. If you want to access the Components collection of a particular application, you need to call ICatalogCollection::GetCollection( ) on the Applications collection interface, passing in the key to the particular application whose Components collection you wish to access.

6.4.3 The ICatalogObject Interface

Every catalog object supports the ICatalogObject interface, allowing you to configure the object's properties. All catalog objects support three predefined read-only properties: Key, Name, and Valid, defined as:

```
[id(2),propget] HRESULT Key([out,retval]VARIANT*
pvarKey);
[id(3),propget] HRESULT Name([out,retval]VARIANT*
pvarName);
[id(5),propget] HRESULT Valid([out,retval]VARIANT_BOOL*
pbValid);
```

The Name property contains the name of the object. For example, if the object is a COM+ application, the name will be the application's name. The Valid property returns TRUE if the object was read successfully from the COM+ Catalog when its containing collection was populated. The `Key` property returns a unique key identifying this object, used to access all the collections associated with that object.

In addition, all catalog objects support, according to their specific type, *named value properties.* These properties are accessible via one read-write property called the `Value` property, defined as:

```
[propget, id(1)] HRESULT Value([in]BSTR bstrPropName,
                              [out,retval]VARIANT*
pvarValue);
[propput, id(1)] HRESULT Value([in]BSTR
bstrPropName,[in]VARIANT varNewValue);
```

Each catalog object (application, component) has a predefined set of named properties and predefined enum values for those properties.

For example, every catalog object in the `Applications` collection represents a COM+ application and has a named value property called `Activation` that controls whether the application should be activated as a library or server application. The predefined enum values for the `Activation` property are `COMAdminActivationInproc` and `COMAdminActivationLocal`.

The `ICatalogObject` interface also supports two not-so-useful helper methods, `IsPropertyReadOnly( )` and `IsPropertyWriteOnly( )`, intended to be used during generic iteration, when you do not know the exact behavior of a property you are accessing.

6.4.4 Using the Catalog Interfaces

You have probably had as much dry theory as you can take, and an example can go a long way to demonstrate the point. Example 6-2 shows many of the points covered so far in this chapter. Suppose you want to programmatically set a COM+ application (called `MyApp`) to be a library COM+ application. Example 6-2 uses Visual Basic to iterate over the `Applications` collection, looking for the `MyApp` COM+ application, and sets its activation mode to a library application.

Example 6-2. Visual Basic example of finding an application and setting its activation mode

```
Dim catalog As ICOMAdminCatalog
Dim applicationCollection As ICatalogCollection
Dim applicationCount As Long
Dim i  As Integer 'Application index
Dim application  As ICatalogObject


Set catalog = New COMAdminCatalog
Set applicationCollection =
catalog.GetCollection("Applications")
Set catalog = Nothing 'You don't need the root any more


'Read the information from the catalog
Call applicationCollection.Populate
applicationCount = applicationCollection.Count( )

For i = 0 To applicationCount - 1
    'Get the current application
    Set application = applicationCollection.Item(i)
    If application.Name = "MyApp" Then
      application.Value("Activation") =
COMAdminActivationInproc
      applicationCollection.SaveChanges
    End If
    Set application = Nothing
    i = i + 1
Next i

Set applicationCollection = Nothing
```

First, create a Catalog root object, the `catalogRoot` object. Then invoke its `GetCollection( )` method, asking for an `ICatalogCollection` interface pointer to the `Applications` collection. Next, release the root object, because it is no longer needed. Then populate the application collection object and find out how many applications you have (the `Count` property). The `for` loop iterates over the applications and gets one application at a time, in the form of an `ICatalogObject` object, using the collection's `Item` property. You then check if the catalog object's name is MyApp. If it is, set its `Activation` named property to the predefined enum value of `COMAdminActivationInproc`. After making the change to the application object, call `SaveChanges( )` on the `Applications` collection object to save the change.

Example 6-3 does the same thing as Example 6-2, except it is written in C++ instead of Visual Basic.

Example 6-3. C++ example of finding an application and setting its activation mode

```
HRESULT hres = S_OK;
ICOMAdminCatalog* pCatalog  = NULL;
ICatalogCollection* pApplicationCollection = NULL;
long nApplicationCount = 0;
int i = 0; //Application index

hres =
::CoCreateInstance(CLSID_COMAdminCatalog,NULL,CLSCTX_ALL,
IID_ICOMAdminCatalog,(void**)&pCatalog);
hres = pCatalog->GetCollection(_bstr_t("Applications"),

(IDispatch**)&pApplicationCollection);
pCatalog->Release( ); //You don't need the root any more

hres = pApplicationCollection->Populate( ); //Read the
information from the catalog
hres = pApplicationCollection-
>get_Count(&nApplicationCount);

for(i=0;i<nApplicationCount;i++)
{
  ICatalogObject* pApplication = NULL;
  //Get the current application
  hres = pApplicationCollection-
>get_Item(i,(IDispatch**)&pApplication);
  _variant_t varAppName;
  _variant_t
varActivation((bool)COMAdminActivationInproc);
  hres = pApplication->get_Name(&varAppName);
  if(_bstr_t("MyApp") == _bstr_t(varAppName))
  {
    long ret = 0;
    hres = pApplication-
>put_Value(_bstr_t("Activation"),varActivation);
    hres = pApplicationCollection->SaveChanges(&ret);
  }
  pApplication->Release( );
}
pApplicationCollection->Release( );
```

A valid question you are probably asking is, "How do I know what the predefined named properties and enum values are for the property I want to configure?" The answer is simple: the Platform SDK documentation (available in the MSDN Library, under *Component Services/COM+ (Component Services)/Reference/COM+ Administration Reference*) contains a comprehensive list of every named property and its corresponding enum values (or data type and range, if applicable).

Another point worth demonstrating with an example is using the `Key` property of a catalog object to access a related collection. Suppose you would like to print to the trace window all the components in all the applications. You would use the `Key` property of every COM+ application to access its `Components` collection. Example 6-4 shows the `TraceTree( )` method that iterates over the `Applications` collection, calling the `TraceComponents( )` method to iterate over an application component collection.

Example 6-4. Tracing all the components in every COM+ application

```
#include "COMadmin.h"
void TraceTree(  )
{
   HRESULT hres = S_OK;
   ICOMAdminCatalog* pCatalog                 = NULL;
   ICatalogCollection* pApplicationCollection = NULL;
   long nApplicationCount = 0;

   hres =
::CoCreateInstance(CLSID_COMAdminCatalog,NULL,CLSCTX_ALL,

IID_ICOMAdminCatalog,(void**)&pCatalog);

   hres = pCatalog-
>GetCollection(_bstr_t("Applications"),

(IDispatch**)&pApplicationCollection);
   pCatalog->Release(  ); //You don't need the root any
more

   //Read the information from the catalog
   hres = pApplicationCollection->Populate(  );
   hres = pApplicationCollection-
>get_Count(&nApplicationCount);

   //Iterate over the Applications collection
   for(int i=0;i<nApplicationCount;i++)
   {
      ICatalogObject* pApplication = NULL;
      ICatalogCollection* pComponentCollection = NULL;
      _variant_t varAppName;
      //Get the current application
      hres = pApplicationCollection-
>get_Item(i,(IDispatch**)&pApplication);
      hres = pApplication->get_Name(&varAppName);

      TRACE("The components in application \"%s\" are:
\n",

(char*)(_bstr_t(varAppName));

TraceComponents(pApplicationCollection,pApplication);
      pApplication->Release(  );
   }
   pApplicationCollection->Release(  );
}

void TraceComponents(ICatalogCollection*
pApplicationCollection,
                     ICatalogObject* pApplication)
{
   HRESULT hres = S_OK;
   ICatalogCollection* pComponentCollection = NULL;
   long nComponentCount = 0;
   _variant_t varAppKey;

   //Get the Component collection for this application.
Need the key first
   hres  = pApplication->get_Key(&varAppKey);
   hres = pApplicationCollection-
>GetCollection(_bstr_t("Components"),

varAppKey,(IDispatch**)&pComponentCollection);

   //Read the information from the catalog
   hres = pComponentCollection->Populate(  );
   hres = pComponentCollection-
>get_Count(&nComponentCount);


   for(int j=0;j<nComponentCount;j++)
   {
      ICatalogObject* pComponent   = NULL;
      _variant_t varCompName;
      //Get the current component
      hres = pComponentCollection-
>get_Item(j,(IDispatch**)&pComponent);
      hres = pComponent->get_Name(&varCompName);
      //Ugly, but works:
      TRACE("  %d. %s \n"
,j+1,(char*)(_bstr_t(varCompName));
      pComponent->Release(  );
   }
   pComponentCollection->Release(  );
}
```

The output from Example 6-4 should look similar to this (depending, of course, on the applications installed on your machine):

```
The components in application "COM+ Utilities" are:
  1. TxCTx.TransactionContext
  2. TxCTx.TransactionContextEx
  3. RemoteHelper.RemoteHelper
  4. QC.Recorder.1
  5. QC.ListenerHelper.1
The components in application "MyApp" are:
  1. MyApp.MyComponent.1
  2. MyObj2.MyObj2.1
  3. Subscriber.MyEvent.1
  4. EventClass.MyEvent.1
The components in application "COM+ QC Dead Letter Queue
Listener" are:
  1. QC.DLQListener.1
The components in application "Logbook" are:
  1. LogBootEvent.LogbookEventClass.1
  2. LogBook.ComLogHTML.1
  3. LogBook.COMLogXML.1
The components in application "System Application" are:
  1. Mts.MtsGrp.1
  2. COMSVCS.TrackerServer
  3. EventPublisher.EventPublisher.1
  4. Catsrv.CatalogServer.1
```

The first part of Example 6-4, the `TraceTree( )` method, creates the root object, gets the top-level `Applications` collection, populates it, and retrieves the number of applications (using the `Count` property). It then iterates over the `Applications` collection, getting one catalog object at a time, tracing its name, and passing it to the `TraceComponents( )` method. The `TraceComponents( )` traces out all the components associated with that application. Note that it is not sufficient to pass to the `TraceComponents( )` method just the application catalog object. You have to pass in as a parameter the `Applications` collection as well. Recall that when you want to access a Collection 2 associated with Object 1 (contained in Collection 1), you get Collection 2 from Collection 1, which contains Object 1. This is why `TraceComponents( )` accepts `pApplicationCollection` as a parameter:

```
void TraceComponents(ICatalogCollection*
pApplicationCollection,
                     ICatalogObject* pApplication)
```

`TraceComponents( )` then calls `get_Key( )` on the application catalog object passed in and, using that key, accesses the application object's `Components` collection. Next, `TraceComponents( )` populates the `Components` collection, gets its count, and iterates over it, tracing one component name at a time.

When writing code as in Example 6-4, which iterates over collections and nested collections, it is very important to name your variables correctly to make your code readable. `ICatalogCollection* pCollection` is a poor variable name, but `ICatalogCollection* pApplicationCollection` is a meaningful and readable name that conveys exactly which collection it is pointing to.

Now you should be getting the feel of how truly generic and extensible the COM+ Catalog programming model really is. The same `ICatalogCollection` interface is used to iterate over every collection, and the same `ICatalogObject` interface is used to configure and access all the parameters in the Catalog, be it an application- or a method-level property.

6.4.5 Saving Changes

When you make changes to a collection (adding or removing catalog objects) or to objects in it (configuring properties), you have to call `ICatalogCollection::SaveChanges( )` to commit them. You can also discard changes you made to a collection, but did not commit yet, by calling `Populate( )` again.
When you call `ICatalogCollection::SaveChanges( )`, all objects and all properties on all the objects are written to the Catalog at once, as an atomic operation. The only problem with this programming model is that the Catalog presents a last-writer-wins behavior—the object is saved in the Catalog precisely the way the last writer configured it. This means that there is a potential for conflicts and contentions between two applications that modify the same data set, because neither has a lock on the items in the Catalog.

6.4.6 Object Properties Interdependencies

Sometimes, a particular value of a catalog object named property depends on the values of other named properties. For example, when the `Transaction` named property of a component is set to the value of `COMAdminTransactionRequired` or `COMAdminTransactionRequiresNew`, the value of the `JustInTimeActivation` named property must be set to `TRUE`. This is no surprise because all transactional components require JITA to be turned on (as well as requiring synchronization).
The COM+ Catalog is aware of all the properties' interdependencies and will enforce consistency whenever it deems it fit. If you try to set a named property in a way that conflicts with another, an error will occur. For example, if you try to turn JITA off on a transactional component (by setting it to `FALSE`), `SaveChanges( )` will fail. One effect of having a smart Catalog is that some properties might be changed for you without you explicitly setting them. For example, if you set the `Transaction` named property to the value of `COMAdminTransactionRequired`, the Catalog turns JITA on and sets the value of the `Synchronization` property to `COMAdminSynchronizationRequired`.

6.5 Features of COMAdminCatalog

There is more to the Catalog root object than providing you with access to the top-level collections. The `ICOMAdminCatalog`interface supports 22 methods, providing you with many useful features that allow you to:

- Connect to the Catalog root object on a remote machine

- Install a new COM+ application
- Export an existing COM+ application
- Start or shut down a COM+ application
- Install components into COM+ applications
- Obtain information regarding event classes
- Start, stop, or refresh load balancing routing (load balancing is not available in standard installations of COM+)
- Check the status of a COM+ service (currently, only load balancing)
- Back up the COM+ Catalog information to a specific file
- Restore the Catalog from a specific file

For example, you often need to programmatically administer a COM+ Catalog on a remote machine, during deployment or for automating remote administration of servers. To do so, you would use the `ICOMAdminCatalog::Connect( )` method, defined as:

```
[id(2)] HRESULT Connect([in]BSTR bstrMachineName,
                        [out,retval]IDispatch**
pRemoteRootCollection)
```

The first parameter to `Connect( )` is the remote machine name, and the second is an out parameter—a pointer to a root collection on the remote machine. After calling `Connect( )`, the `ICOMAdminCatalog` you are holding starts affecting the remote machine to which you have connected—calls made on its methods administer the remote machine. You can also use the `pRemoteRootCollection` parameter to gain access to remote top-level collections, as shown in Example 6-5.

Example 6-5. Accessing a top-level catalog collection on a remote machine

```
HRESULT hres = S_OK;
ICOMAdminCatalog*   pCatalog              = NULL;
ICatalogCollection* pRemoteAppCollection  = NULL;
ICatalogCollection* pRemoteRootCollection = NULL;

//Creating a local catalog
hres =
::CoCreateInstance(CLSID_COMAdminCatalog,NULL,CLSCTX_ALL,

IID_ICOMAdminCatalog,(void**)&pCatalog);

//Connecting to the remote machine
hres = pCatalog->Connect(_bstr_t("RemoteMachineName"),

(IDispatch**)&pRemoteRootCollection);

pCatalog->Release( );///No need for it anymore
```

```
_variant_t varKey("");//Key value will be ignored

//Getting the "Applications" collection on the remote
machine
hres = pRemoteRootCollection-
>GetCollection(_bstr_t("Applications"),varKey,

(IDispatch**)&pRemoteAppCollection);

pRemoteRootCollection->Release( );//No need for the
remote root collection anymore

/* use pRemoteAppCollection */

pRemoteAppCollection->Release( );
```

Another example of what you can do with the root object is shutting down and starting up COM+ applications. The `ICOMAdminCatalog` interface supports the `StartApplication( )` and `ShutdownApplication( )` methods, defined as:

```
[id(16)] HRESULT StartApplication(BSTR strAppName);
[id(8)]  HRESULT ShutdownApplication(BSTR strAppName);
```

Starting up an application programmatically is helpful in the case of queued components (you will see why in Chapter 8), and shutting down COM+ applications is extremely useful during development. When you are doing a test-debug-fix-build-retest cycle, you often discover a problem that you can fix on the spot. However, you cannot rebuild your components as long as the application that hosts them is running because the application maintains a lock on the DLL. A COM+ application may be running even when idle (the default is three minutes), so you have to shut down the application using the Component Services Explorer. After a while, this becomes very annoying. The situation is even worse if you have a number of interacting COM+ applications and you have to shut them all down—for example if you want to change a header file, a lib, or a component they all use.

Replicating the COM+ Catalog

If your product consists of more than one COM+ application, you may want to actually clone the entire COM+ Catalog on a machine where the product is installed and use the clone as an installation. COM+ allows you to replicate all COM+ settings from a giving source computer to one or more target computers, using a utility called `COMREPL`. `COMREPL` is typically used to replicate a master configuration and deploy it on a set of identically configured computers. Another potential use for `COMREPL` is for product configuration management purposes.

COMREPL is a crude command line-driven utility:
COMREPL <source computer name> <target computers list>
All COM+ applications on the master computer are replicated to the target computers, except the COM+ preinstalled applications. In addition, all COM+ applications previously installed on the target computers will be deleted as part of the replication process.

So how about building a utility that uses ICOMAdminCatalog::ShutdownApplication( ) to shut down the application specified on the command line—or all of the COM+ applications on your machine, if no application name was specified? I call this utility Nuke'm, and I even have a special icon on my Visual Studio toolbar that I click before every build, just to purge all the running applications from my machine and start a fresh build and test cycle. Nuke'm contains a light C++ wrapper class around the ICOMAdminCatalog interface, called CCatalogAdmin. Example 6-6 shows its Shutdown( ) method, which shuts down the specified application and, if none is specified, shuts down all the COM+ applications.

Example 6-6. The CCatalogAdmin::ShutDown( ) method

```
HRESULT CCatalogAdmin::ShutDown(BSTR bstrAppName)
{
   //m_pCatalog is a member of the class, initialized in
the constructor
   if(_bstr_t(bstrAppName) != _bstr_t(""))
   {
      return  m_pCatalog-
>ShutdownApplication(bstrAppName);
   }
   else//Shut down all the applications
   {
      HRESULT hres = S_OK;
      ICatalogObject* pApplication  = NULL;
      ICatalogCollection* pApplicationCollection = NULL;
      long nApplicationCount = 0;
      int i = 0;//Application index

      //Get the application collection
     hres = m_pCatalog-
>GetCollection(_bstr_t("Applications"),

(IDispatch**)&pApplicationCollection);

      hres = pApplicationCollection->Populate(  );
      hres = pApplicationCollection-
>get_Count(&nApplicationCount);
```

```
      for(i=0;i<nApplicationCount;i++)
      {
         //Get the current application
         hres = pApplicationCollection->get_Item(i,
(IDispatch**)&pDispTemp);

         _variant_t varName;
         hres = pApplication->get_Name(&varName);
         _bstr_t bstrName(varName);

       //No point in killing the system app,
       //since it will start up again immediately
         if(bstrName != _bstr_t("System Application"))
         {
            hres = m_pCatalog-
>ShutdownApplication(bstrName);
         }
         pApplication->Release(  );
      }
      pApplicationCollection->Release(  );
      return hres;
   }
}
```

The Nuke'm utility is available from this book's web site, http://www.oreilly.com/catalog/comdotnetsvs/.

6.6 The COM+ Catalog and Transactions

The COM+ Catalog is a resource manager. When a component that takes part in a transaction tries to access the Catalog, the Catalog auto-enlists in that transaction. As a result, all the configuration changes made within the scope of that transaction will be committed or aborted as one atomic operation, even across multiple catalogs on multiple machines, according to the transaction success. The main advantage of having the COM+ Catalog take part in your transactions is that it enormously simplifies deployment on multiple machines. Imagine a situation in which you write an elaborate installation script that tries to access and install your product on multiple machines. The problem is that almost anything in a distributed installation scenario can go wrong—from network failures to security to disk space. Because all the installation attempts are scoped under one transaction, you can guarantee that all server machines are left with identical configurations—either the installation succeeded on all of them, or the changes were rolled back and the servers are left just as they were before you tried to install the product.

Another benefit of having the Catalog as a resource manager is dealing with potential contentions and conflicts between two different applications that try to access and modify the Catalog at the same time. To ensure the transaction's isolation, when one transaction makes a change to the Catalog, the Catalog will block all writers from other transactions until the current transaction commits or aborts. (COM+ will abort the transaction if a deadlock situation exists because of the blocking.) While a transaction modifies the Catalog, readers from within that transaction will read the data as if it were committed. Readers from outside the transaction will not be blocked, and the data they see will not reflect any interim changes made within the first transaction until that transaction actually commits. You should avoid starting a new COM+ application (either programmatically or manually via the Component Services Explorer) that relies on information that is not yet committed.
One last point regarding transactions and the COM+ Catalog: you can programmatically invoke calls that access the filesystem, such as exporting a COM+ application. The problem is that the filesystem and the Windows Installer do not participate in transactions. If your transaction aborts, you will have to roll back those changes manually to maintain consistency.

6.7 Summary

Programming the COM+ Catalog is nothing more than understanding the Catalog programming model and navigating down the Catalog structure, using the Component Services Explorer or the Catalog structure diagrams in this chapter as reference guide. This chapter focused on the Catalog structure, not on the semantics of the items it contains. Although the Catalog interfaces were designed for scripting languages, you can access them from C++ as well, and the resulting code is just as concise. Some COM+ services features are available only by accessing the Catalog programmatically (in particular, some features of COM+ Events, discussed in Chapter 9), so knowing how to work with the Catalog is an essential skill. Furthermore, automating mundane and repetitive development and deployment tasks by programming directly against the COM+ Catalog is fairly easy.

Chapter 7. COM+ Security

Perhaps nothing epitomizes the differences between developing a distributed enterprise-wide system using COM+ and developing one using DCOM more than the COM+ security service. DCOM security is notorious for being complex and hard to learn. Even though DCOM uses a simple and elegant security programming and configuration model, the sheer volume of technical details and the inherent difficulty of distributed systems security puts DCOM security outside the reach of many developers.

COM+ makes using security enjoyable by providing an easy-to-use administrative security infrastructure. COM+ security is based on an intuitive new security concept called *role-based security.* Role-based security greatly simplifies the management and configuration of your application's security. Of all component services provided by COM+, security is my favorite.

COM+ security makes it possible for you to leave all security-related functionality outside the scope of your components and configure security administratively. *Roles* are used for access control, and *declarative attributes* are used for the remaining security settings. If the administrative configurations are too coarse for your particular needs and you still want to have programmatic control over security, COM+ provides an easy-to-use programmatic way to fine-tune security. In fact, COM+ security solves classic distributed computing problems that are difficult and would require much work to solve on your own. Even with a single-machine application, COM+ security provides elegant solutions for administration and configuration issues.

This chapter covers basic security concepts, but it avoids (as much as possible) the gory details of low-level security manipulation and COM+ security implementation. Instead, I'll focus on how best to use the security service, what the available options are, the tradeoffs between them, and their configuration pitfalls.

7.1 The Need for Security

Who needs security? You do. Almost nobody today develops a standalone, single-machine, self-contained application. Applications today are distributed between multiple machines. Some applications have a user interface; others execute business logic and interact with other applications. Your database is probably on a separate set of machines altogether. The word "security" is intrinsic to the word "distributed"—meaning that the moment you distribute your application, security raises its head.

Security provides ways to verify that a particular user has sufficient credentials to perform an operation. Security is the way you verify that the users are who they say they are. Security is the way you protect your system from innocent user mistakes and malicious attacks. For example, imagine a hospital patient information system. In this system, not all users on all terminals are created equal. Only doctors can sign a death certificate or change a dose of medicine. Nurses can update patient parameters, such as temperature or the last time the patient took medicine. Hospital clerks can view some information and bill the patient’s insurance company. However, a clerk should not be allowed to alter anything considered medical information, not even accidentally. A security infrastructure provides an easy way to configure these credentials and access controls. When a doctor logs on at a nurse’s station, you want to give the doctor proper access, even though the access is from the nurse’s station. You should protect the privacy of the patient information so malicious parties—on the inside or outside—cannot gain access to it. You want to be able to easily change who is allowed to do what and avoid hardcoding security policies in your application. As the system and the domain change (new hospital regulations or new users), you want to reconfigure the system security without recoding it.
Security in a modern system is not an afterthought. You must design security into your COM+ application and components from day one, much the same way you design concurrency and threading models, factor out your interfaces, and allocate interfaces to components. If you don't, at best your application will not work. At worst, you will introduce security breaches into your system, allowing critical application logic to go astray and face data corruption or inconsistency. Essentially, lack of security is a failure to deliver the robust system your customer pays for. When dealing with security, you should always assume that somebody will eventually find and take advantage of a security hole.

7.2 Basic Security Terms

To make the most of the security configurations COM+ has to offer, you need to be familiar with a few basic terms and concepts. The rest of this chapter makes frequent use of these terms.

7.2.1 Security Identity

A *security identity* is a valid account used to identify a user. The account can be local or an account on a domain server. Every COM+ entity, be it a client or an object, must have an identity associated with it so that COM+ can determine what that entity is capable of accessing. In Windows, all objects in the same process share the same identity, unless they make an explicit attempt to assume a different identity. You can configure a COM+ server application to always run under a particular identity or to run under the identity of the user who is currently logged on that Windows station. Objects from a COM+ library application run under the identity of the hosting process by default.

7.2.2 Authentication

Authentication has two facets. The first is the process by which COM+ verifies that the callers are who they claim to be. The second is the process by which COM+ ensures the integrity of the data sent by the callers. COM+ authentication relies on the underlying security provider—in most cases Windows 2000 built-in security.
In the Windows default security provider, the *challenge/response* protocol is used to authenticate the caller's identity. Given that all callers must have a security identity, if the callers are who they say they are, then they must know the account password. That password is also known to the domain server. The security provider does not want to ask the callers directly for their passwords because a malicious third party can sniff the network to discover the password. Instead, to authenticate the callers, the security provider encodes a random block of data with the account password and sends it to the callers, asking them to decode the encrypted block using the password and send the result back. This process is the *challenge*. If the returned block, the *response*, is the same as the original unencrypted block, then the callers are authenticated.
Authenticating caller identity is only one problem. The other problem is that data passed in a method call can be intercepted, copied, altered, or corrupted by a malicious third party. Under COM+, both the caller and the object have a range of choices to determine how secure the connection between them should be.
To authenticate data integrity, COM+ can use one of two techniques: it can append a checksum to every network packet, making sure that the data is not tampered with during transport, or it can encrypt all information in the packet.
Both kinds of authentication (identity and data integrity) are, in most cases, completely transparent to both the caller and the object and done automatically by COM+. However, there is a clear tradeoff between security and performance (when and to what extent to authenticate), and it is up to you to choose and configure the proper authentication level for your application.

7.2.3 Authorization

Authorization is the process of determining what the caller is allowed to access in the system. Authorization is also called *access* *control.* COM+ uses role-based security (discussed in the following section) to let you define access control at the component, interface, and method levels. Access control is used to protect objects and resources against unauthorized access by clients. If a user who is not granted access to a component tries to invoke a method on that component, the method invocation fails with the error code `E_ACCESSDENIED` ("Permission Denied" in Visual Basic). You configure access control administratively using the Component Services Explorer. Programmatically, you can still fine-tune access and execution of a method based on the caller's identity and other information such as the method parameters and object state. Note that authorization is not related to authentication. Authorization assumes that the caller is already authenticated and is only concerned with whether the caller can access this object. It is not concerned with whether the caller is really who he or she claims to be.

7.2.4 Launch Security

Launch security controls which users are allowed to create a new object in a new process. Unlike DCOM, COM+ does not provide a dedicated way to control launch security. This is done intentionally to avoid a common DCOM security pitfall—allowing a user to launch a process, but forgetting to grant the user access to the objects inside! As a result, the user could call `CoCreateInstance( )` to launch the process, but would be denied access to methods, including being unable to call `Release( )` on the object. The process is ultimately orphaned, and the user has to shut it down manually or rely on COM garbage collection to eventually shut the process down. In COM+, even if the client is not granted access to the object, (but is a member of at least one role defined for the application), the client can still launch a new process with a new object inside and can call the `IUnknown` methods on the object, including `Release( )`. The client cannot access methods on any other interface, however.

7.2.5 Impersonation

Authorization and authentication protect the object from being accessed by unauthorized and unauthenticated users. This protection ensures that when an object is asked to perform an operation, the invoking client has permission to access the system and the call was not initiated by an adversary client. However, how should the client be protected from malicious objects? What prevents the server from assuming the client's identity and credentials and causing harm? Is the server even allowed to learn the identity of the calling client? By setting the *impersonation* level, COM+ lets callers indicate what they allow objects to do with their security identity. The impersonation level indicates the degree to which the server can impersonate the calling client. Setting the impersonation level can be done administratively and programmatically on the client side; attempting to impersonate the client can only be done programmatically by the server.

7.3 Role-Based Security

The cornerstone of COM+ access control is role-based security. A *role* is a symbolic category of users who share the same security privileges. When you assign a role to an application resource, you grant access to that resource to whoever is a member of that role.

7.3.1 Configuring Role-Based Security

The best way to explain role-based security is by demonstration. Suppose you have a COM+ banking application. The application contains one component, the bank component. The bank component supports two interfaces that allow users to manage bank accounts and loans, defined as:

```
interface IAccountsManager : IUnknown
{
   HRESULT TransferMoney([in]int nSum,[in]DWORD
dwAccountSrc,
                        [in]DWORD dwAccountDest);
   HRESULT OpenAccount([out,retval]DWORD* pdwAccount);
   HRESULT CloseAccount([in]DWORD dwAccount);
   HRESULT GetBalance([in]DWORD
dwAccount,[out,retval]int* pnBalance);
};
interface ILoansManager : IUnknown
{
   HRESULT Apply([in]DWORD dwAccount,[out,retval]BOOL*
pbApproved);
   HRESULT CalcPayment([in]DWORD dwSum,[out,retval]DWORD*
pdwPayment);
   HRESULT MakePayment([in]DWORD dwAccount,[in]DWORD
dwSum);
};
```

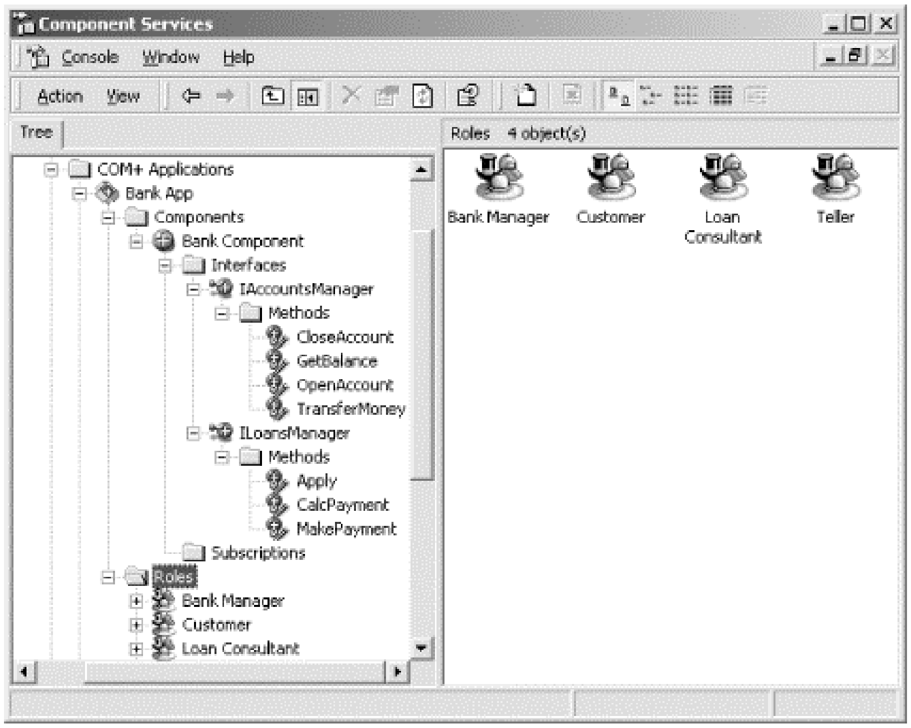
During the requirements-gathering phase of the product development, you discovered that not every user of the application should be able to access every method. In fact, there are four kinds of users:

- The bank manager, the most powerful user, can access all methods on all interfaces of the component.

- The bank teller can access all methods of the `IAccountsManager` interface, but is not authorized to deal with loans. In fact, the application is required to prevent a teller from accessing any `ILoansManager` interface method.
- Similarly, the loan consultant can access any method of the `ILoansManager` interface, but a consultant is never trained to be a teller and may not access any `IAccountsManager` interface method.
- A bank customer can access some of the methods on both interfaces. A customer can transfer funds between accounts and find the balance on a specified account. However, a customer cannot open a new account or close an existing one. The customer can make a loan payment, but cannot apply for a loan or calculate the payments.

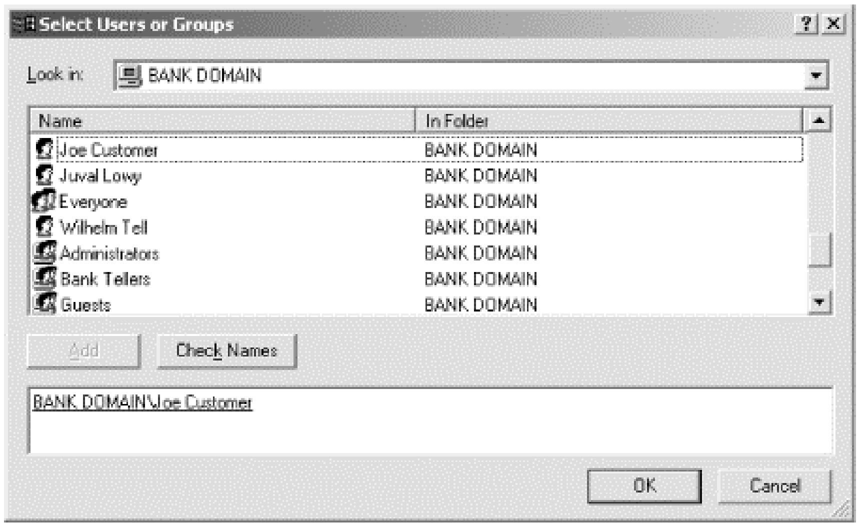
If you were to enforce this set of security requirements on your own, you would face an implementation nightmare. You would have to manage a list of who is allowed to access what and tightly couple the objects to the security policy. The objects would have to verify who the caller is and whether the caller has the right credentials to access them. The resulting solution would be fragile. Imagine the work you would have to do if these requirements were to change. Fortunately, COM+ makes managing such a security access policy easy. After importing the bank component into a COM+ application (be it a server or a library application), you need to define the appropriate roles for this application. Every COM+ application has a folder called *Roles*. Expand the *Roles* folder, right click on it, and select New from the context menu. Type **`Bank Manager`** into the dialog box that comes up and click OK. In the *Roles* folder, you should see a new item called Bank Manager. Add the rest of the roles: **`Customer`** , **`Teller`** , and **`Loans Consultant`** . The application should look like Figure 7-1.

Figure 7-1. The Roles folder of the bank application



You can now add users to each role. You can add any user with an account on the machine or the domain. Every role has a *Users* folder under which you add registered users from your domain or the machine local users. For example, navigate to the *Users* folder of the Customer role, right-click the *Users* folder, and select New from the Context menu. In the dialog box, select the users who are part of the Customer role, such as Joe Customer (see Figure 7-2). You can populate this role and the remaining roles in the bank application with their users.

Figure 7-2. Populating a role with users

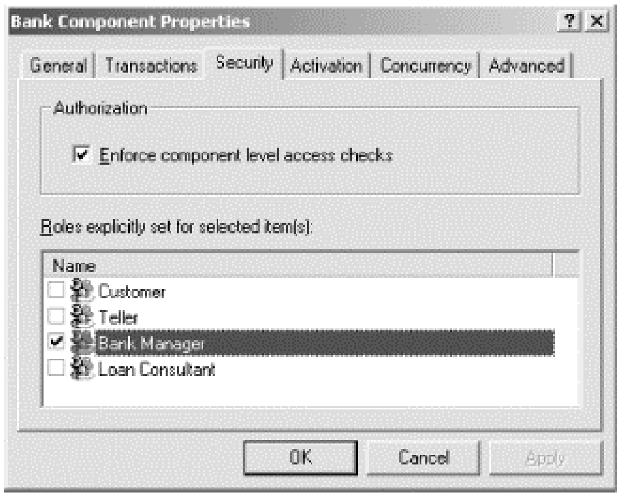


Name	In Folder
Joe Customer	BANK DOMAIN
Juval Lowy	BANK DOMAIN
Everyone	BANK DOMAIN
Wilhelm Tell	BANK DOMAIN
Administrators	BANK DOMAIN
Bank Tellers	BANK DOMAIN
Guests	BANK DOMAIN

BANK DOMAIN\Joe Customer

The next step is to grant access to components, interfaces, and methods for the various roles in the application, according to the bank application requirements. Display the bank component properties page and select the Security tab. The tab contains the list of all roles defined for this application. Check the Manager role to allow a manager access to all interfaces and methods on this component (see Figure 7-3). When you select a role at the component level, that role can access all interfaces and methods of that component. Make sure that the "Enforce component level access check" checkbox under Authorization is selected. This checkbox, your component access security switch, instructs COM+ to verify participation in roles before accessing this component.

Figure 7-3. Selecting a role at the component level


Bank Component Properties

General | Transactions | Security | Activation | Concurrency | Advanced

Authorization

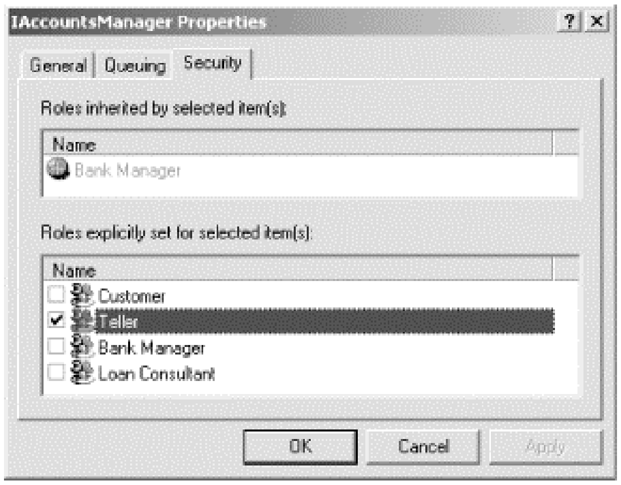
[✓] Enforce component level access checks

Roles explicitly set for selected item(s):

Name
[ ] Customer
[ ] Teller
[✓] Bank Manager
[ ] Loan Consultant

Next, configure security at the interface level. Display the `IAccountsManager` interface properties page, and select the Security tab. Select the Teller role to grant access to all methods in this interface to any member of the Teller role (see Figure 7-4). The upper portion of the interface security tab contains *inherited roles* — roles that were granted access at the component level, and thus access to this interface as well. Even if the Bank Manager role is not checked at the `IAccountsManager` interface level, that role can still access the interface.

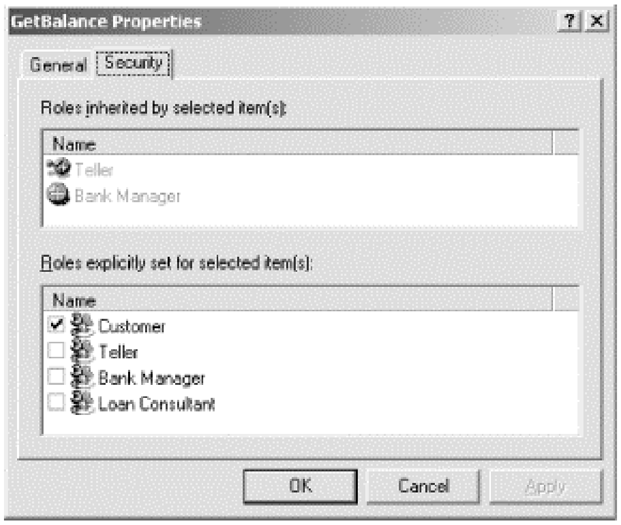
Figure 7-4. Granting access to a role at the interface level


IAccountsManager Properties
General
Queuing
Security
Roles inherited by selected item(s):
Name
Bank Manager
Roles explicitly set for selected item(s):
Name
Customer
Teller
Bank Manager
Loan Consultant
OK
Cancel
Apply

Similarly, configure the ILoansManager interface to grant access to the Loans Consultant role. The Bank Manager should also be inherited in that interface. Note that the Loans Consultant cannot access any method on the IAccountsManager interface, just as the requirements stipulate.

Finally, you can configure access rights at the method level. A customer should be able to invoke the GetBalance( ) and TransferMoney( ) methods on the IAccountsManager interface, and the MakePayment( ) method on the ILoansManager interface, but no other methods. Granting access at the method level is similar to granting it at the interface or component level. For example, to configure the GetBalance( ) method, display that method's Properties page, select its Security tab and check the Customer role (see Figure 7-5). The method's Security tab shows inherited roles from the interface and component levels. COM+ displays roles inherited from the component level with a component icon; it shows roles inherited from the interface level with an interface icon.

Figure 7-5. Granting access to a role at the method level


GetBalance Properties

General | Security

Roles inherited by selected item(s):

Name
Teller
Bank Manager

Roles explicitly set for selected item(s):

Name
✓ Customer
☐ Teller
☐ Bank Manager
☐ Loan Consultant

Because of the inherited nature of roles, you can deduce a simple guideline for configuring roles: put the more powerful roles upstream and the more restricted roles downstream.

7.3.2 Role-Based Security Benefits

For all practical purposes, COM+ role-based access control gives you ultimate flexibility with zero coding. It gives you this flexibility because access control at the method level is usually granular enough. Role-based security offers a scalable solution that does not depend on the number of system users. Without it, you would have to assign access rights for all objects and resources manually, and in some cases you would have to impersonate users to find out whether they have the right credentials. (In Section 7.8, you will see how an object can impersonate a caller.) Configurable role-based security is an extensible solution that makes it easy to modify a security policy. Like any other requirement, your application's security requirements are likely to change and evolve over time, but now you have the right tool to handle it productively.
Role-based access control is not limited to configurations made with the Component Services Explorer. You can build more granular security policies programmatically if you need to, using role-based security as a supporting platform.

7.3.3 Designing Role-Based Security

Roles map nicely to terminology from your application's domain. During the requirements analysis phase, you should aspire to discern user roles and privileges, in addition to discovering interfaces and classes. Focus your efforts on discovering differences in the roles users play that distinguish them from one another, rather than placing explicit permissions on each object in the system. As you saw in the bank example, roles work very well when you need to characterize groups of users based on what actions those users can perform. However, roles don't work well in a couple of cases. First, they don't work well when access decisions rest on the identity of a particular user: for example, if only the bank teller Mary Smiling is allowed to open an account. Second, they don't work well when access decisions rest on special information regarding the nature of a particular piece of data: for example, when bank customers cannot access accounts outside the country. Role-based security is a service that protects access to middle-tier objects. Middle-tier objects should be written to handle any client and access any data. Basing your object behavior on particular user identities does not scale. Forcing your objects to know intimate details about the data does not scale well either. Each security mechanism has its limitations—if your application requires you to implement this sort of behavior, you may want to look at other options, such performing the security access checks at the database itself.

When designing effective roles, try to avoid a very intricate role-based policy. A policy with many roles that allocates users to multiple roles may be too complicated. Role-based security should be a straightforward solution with crisp distinctions between roles. Avoid defining roles with ambiguous membership criteria. The simpler the solution, the more robust and maintainable it will be. Your application administrator should be able to map users to roles instantly. Use meaningful, self-describing names for roles, borrowing as much as possible from the application domain's terms and vocabulary. For example, Super User is a bad role name, whereas Bank Manager is a good name (even though your application would function just fine with the former).

Occasionally, you will be tempted to model a real-life situation and define numerous roles. Maybe different branches of the bank have different policies describing what a teller can do. Try to collapse roles as much as possible. You can do this either by refactoring your interfaces (deciding what methods will be on what interface and which component supports which interface) or by defining new interfaces and components. Breaking the system into more granular COM+ applications, each with its own small set of roles, is another design solution used to cope with numerous roles. This solution would probably be a better modeling of the system in other respects as well.

Avoiding numerous roles also improves performance. On each call, COM+ must scan the list of roles to find out whether the caller is a member of a role that is granted access.

Roles are defined at the application level, but they are actually part of every component's design. If you write a standalone COM+

component that will be placed in COM+ application managed by someone else, you need to have in your documentation explicit instructions describing how to configure security for the hosting application. You need to document that your component needs its access control turned on for this application, the required authentication level, the roles that should be defined for this application, and the criteria that should be used to allocate users for your roles. You need to stipulate which methods and interfaces each role should be granted access to and which roles are granted access to the entire component.

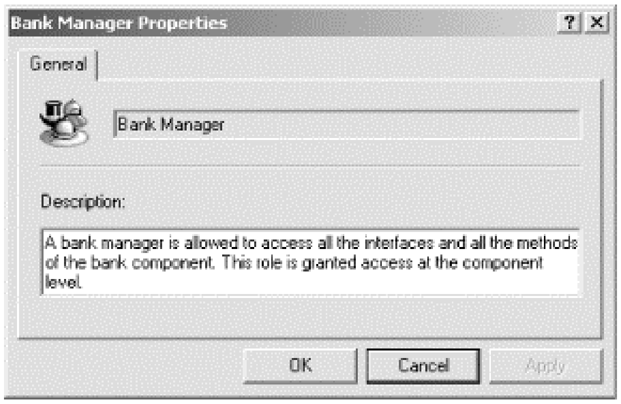
7.3.4 Deploying and Administering Role-Based Security

Roles are an integral part of your design, but allocation of users to roles is part of your application deployment. The application administrator should make the final allocation of users to roles at the customer site. Because you need to make the administrator's job as easy as possible, your application should already have predefined roles, and the administrator should only need to allocate users to roles. When adding users to roles, populating the roles with Windows 2000 user groups instead of individual users is wise. Groups also appear on the same list as users, such as in Figure 7-2, in the Bank Tellers group. By assigning groups to roles, the application is automatically configured to handle the new user correctly when a new user is added to a domain user group. The same is true when a user is removed from a Windows user group or removed from one group and added to another (for example, when Mary Smiling is promoted to a bank manager position). When you assign groups to roles, your application reacts transparently to normal events in the application domain.

> If you target international markets, you should localize your roles and have them translated into the local language. In many cases, application administrators will be local hires on the foreign market, and properly translated roles can make a world of difference.

When providing the best support for your application administrator, you should clearly document the role-based policy you design, whether or not role membership is obvious to you. In particular, use the description field available for each role, as shown in Figure 7-6. The description should be concise. If you cannot describe who should belong to the role in three lines, the role is probably too complex.

Figure 7-6. The Description field on the role properties page


Bank Manager Properties

General

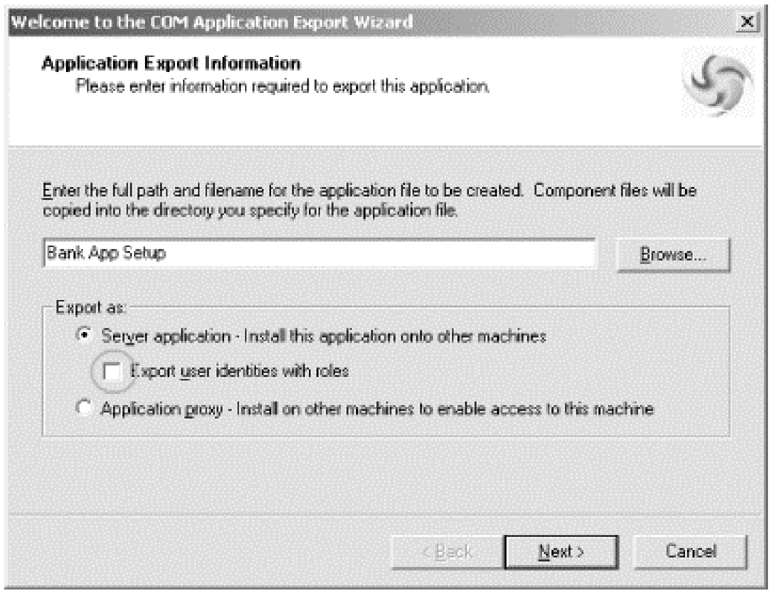
Bank Manager

Description:

A bank manager is allowed to access all the interfaces and all the methods of the bank component. This role is granted access at the component level.

Building a helper administrative utility to add users to roles programmatically, using the COM+ Catalog's interfaces and components, may also be worthwhile; it saves the application administrator the trouble of learning how to use the Component Services Explorer. The utility should present to the administrator a familiar user interface, preferably the same user interface standard as the application itself. The utility should display the users selection dialog box to the administrator and add the selected users to the appropriate roles. When you export a COM+ application, the Application Export Wizard gives you the option of exporting the user identities with the roles (see Figure 7-7)

Figure 7-7. You should usually avoid exporting user identities with roles


Welcome to the COM Application Export Wizard

Application Export Information

Please enter information required to export this application.

Enter the full path and filename for the application file to be created. Component files will be copied into the directory you specify for the application file.

Bank App Setup

Export as:

Server application - Install this application onto other machines [✓]

Export user identities with roles [ ]

Application proxy - Install on other machines to enable access to this machine [ ]

This option should only be used by the application administrator when making cloned installations at a particular site, from one machine to another. Remember that roles are part of the design, while allocation of users to roles is part of deployment. In fact, exporting user information from one deployment site to another may constitute a security breach. Most customers would not like a list of their employees, their usernames, and the roles they play in the organization available at large, let alone at some other company's site. As a developer, "export user identities with roles" is of little use to you.

7.4 Securing a Server Application

Controlling access to your components via role-based security is all fine and well, but there is more to security than just access control. You must still set the security identity for your application and set the authentication and impersonation levels. Configuring security for a server application is different from that of a library application, justifying each application type in a separate section.
When designing and configuring a server application security, you need to do the following:

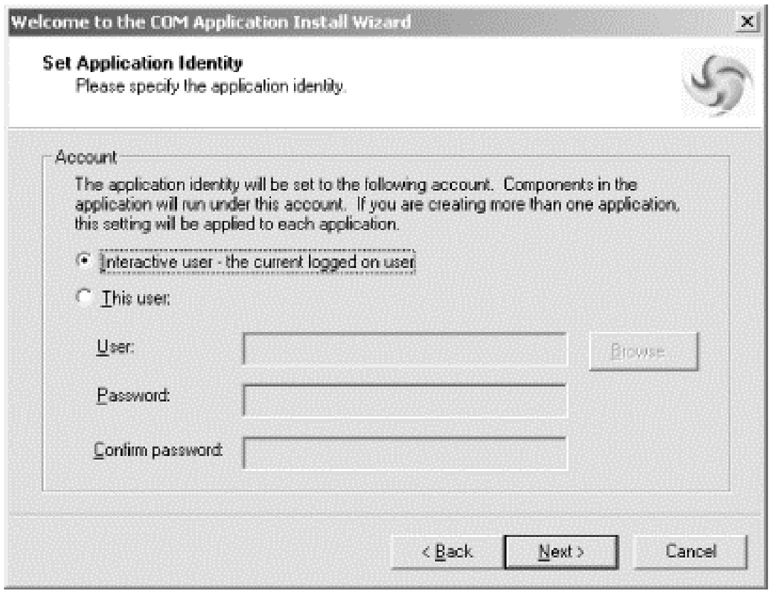
- Decide on the security identity under which the server application executes.
- Decide what authorization (access control) the server application requires—how granular access control should be.
- Decide at what authentication level to authenticate incoming calls.
- Decide at what impersonation level you grant objects in other applications when this server application is the client of those objects.
- Configure your server application security.

The following sections discuss these action items in depth.

7.4.1 Configuring the Server Application Identity

When you invoke the Application Install Wizard and use it to create a new server application, the Wizard presents you with a dialog box that lets you set the security identity of the server application. Setting the security identity determines what user account all components in that application will run under, which dictates credentials, privileges, and access rights (see Figure 7-8). You may either run the application as the interactive user (useful during debugging) or as a designated user (for deployment).

Figure 7-8. Selecting an identity for a new server application


Welcome to the COM Application Install Wizard
Set Application Identity
Please specify the application identity.
Account

The application identity will be set to the following account. Components in the application will run under this account. If you are creating more than one application, this setting will be applied to each application.

(●) Interactive user - the current logged on user

( ) This user:

User:

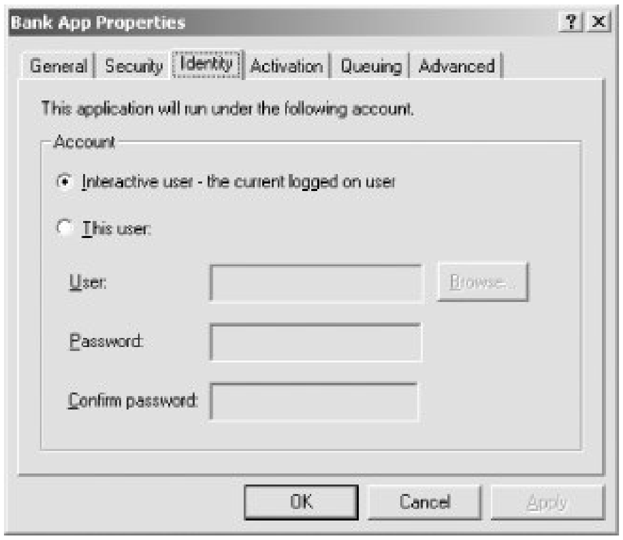
Password:

Confirm password:

Browse...
< Back
Next >
Cancel

You can always set a different identity later on (and you usually will) by bringing up the application properties page and selecting the Identity tab (see Figure 7-9).

Figure 7-9. Selecting an identity for an existing server application



This application will run under the following account.

Account

(●) Interactive user - the current logged on user

( ) This user:

User:

Password:

Confirm password:

When Object A is created in the application, the application security identity controls everything Object A is allowed to access and do. If Object A tries to access another object (Object B) in another application, and Object B is configured to use role-based security, COM+ uses the security identity of Object A to determine whether to grant access to Object B. The security identity of Object A has to belong to at least one role that Object B allows access to. But there is more to an object's identity than role-based security: accessing the filesystem, accessing Win32 handles, installing new components, accessing the COM+ Catalog, modifying the Registry,

remote calls, and so on, are all limited by the privileges of the security identity.
To make an educated decision on selecting the right identity for your objects, you need to know the term *Windows station.* In Windows, every user, or more precisely, every security identity, gets to run in its own station—it has its own copy of the clipboard, global atoms table, desktop objects, a keyboard, a mouse, and a display device. Each logged-on user is provided with a new Windows station. Obviously, only the Windows station associated with the currently interactive user can actually display a user interface. If a component is set to run under a designated security identity and that identity is different from that of the interactive user, it is placed in its own Windows station.
When you configure your server application identity to run under the account of the interactive user, the application shares the interactive Windows station with that user. This option has the clear benefit of being able to interact with the user. However, it also has severe limitations: what should COM+ do if no user is logged on and an activation request from another machine tries to launch the application? In this case, COM+ refuses to launch the application. If the interactive user logs off, COM+ also terminates the application.
The second option COM+ provides for configuring a server application's identity is to run under a specific designated identity. The application is placed in its own Windows station. All subsequent instantiations of new components from that application share that dedicated windows station and identity credentials. The component in the application cannot have a user interface because their Windows station cannot interact with the user. However, for a middle-tier component, a user interface is not necessary anyway; all user interaction is performed at the presentation tier. You can still redirect message boxes to the interactive Windows station, using the message box type attribute `MB_DEFAULT_DESKTOP_ONLY`. This redirection is done by design for debug purposes and is available for message boxes only.

Running as Activator

The architects of COM+ actually had, in theory, a third option for a server application security identity. That third option is to run under the identity of the launching user. This option is available under classic DCOM (in fact, it is the default for DCOM). However, it has a few critical limitations: if COM+ were to create a new Windows station for every new activation request coming from a different identity, the system would run out of resources very quickly because a Windows station is extremely expensive to create and maintain. As a result, this option does not scale well at all. Another limitation is the potential for having objects from the same application running in different processes because every Windows station has its own initial process. This potential could violate design decisions—you may have wanted all your objects in one process because they may need to share event handles or some other process-wide resource. Given these limitations, you can understand why the COM+ architects chose not to include the option to launch the application under the identity of the launching user.

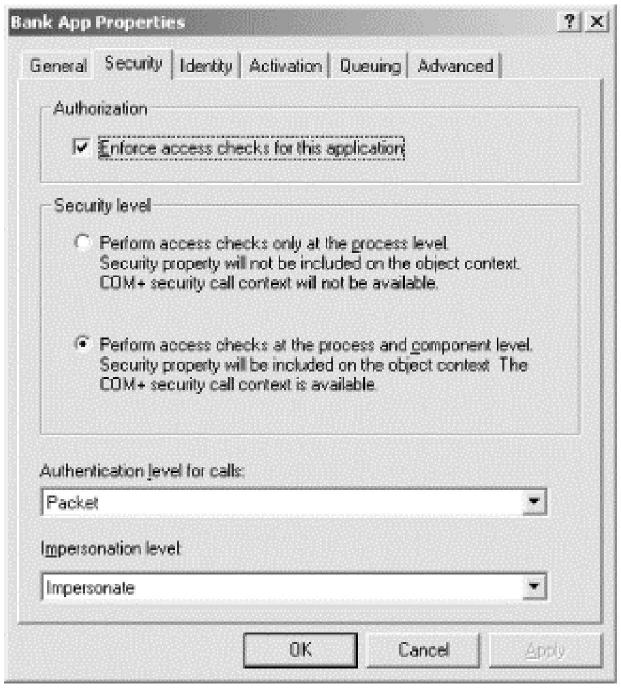
So, which of the two options should you choose? Running as the interactive user has a distinct advantage during debugging sessions, because you can use a debugger to trace the execution of your components. In addition, during a debug session, the developer is logged on to his machine, so COM+ activates the application easily. Running as a designated user is more useful for deployment purposes. It frees you from needing a user logged on to the server machines when your application is running. If you configure more than one application to run under the same designated user account, you also conserve system resources because all components from those applications share the same Windows station. Running under a specific identity has a few more advantages:

- Because an object can perform operations on behalf of arbitrary users, limiting the object's capabilities is often necessary. By assigning the object a less privileged identity, you limit the potential harm malicious callers can do after being granted access the object (the interactive user may have unlimited administrator power, and that could be very dangerous indeed).
- Internet clients calling into your application have no identity at all and are anonymous in most cases. You can now assign a specific identity to the objects that carry out a request on behalf of Internet clients.

7.4.2 Enabling Authorization

The properties page of each COM+ server application includes a Security tab. The security tab is where you set the rest of the security properties for your application. There are four settings on this tab, each discussed in the following sections. At the top of the tab (see Figure 7-10), you will find the authorization checkbox.

Figure 7-10. A server application Security tab



Bank App Properties

General | Security | Identity | Activation | Queuing | Advanced

Authorization

[✓] Enforce access checks for this application

Security level

( ) Perform access checks only at the process level.
Security property will not be included on the object context.
COM+ security call context will not be available.

(●) Perform access checks at the process and component level.
Security property will be included on the object context. The
COM+ security call context is available.

Authentication level for calls: Packet

Impersonation level: Impersonate

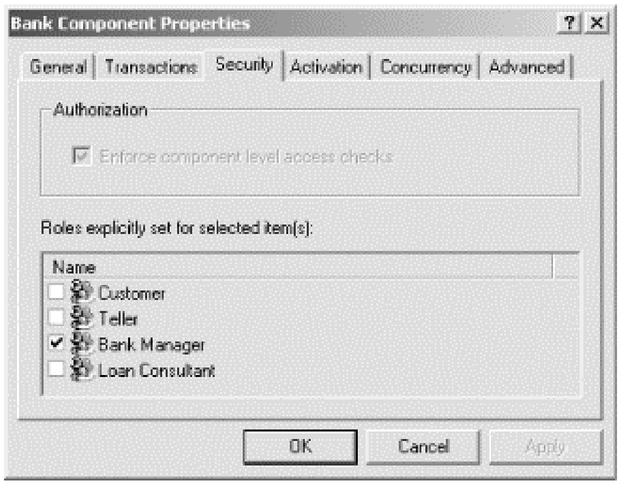
OK | Cancel | Apply

The authorization checkbox is the access security master switch for the *application* (The component's developer still has to enable the component-level authorization on a component by component basis, as discussed previously; see Figure 7-3). When you install a new COM+ application, either a library or a server application, the default setting for this switch is off. You must turn on authorization yourself by checking the checkbox to enable role-based security for your application. When authorization is enabled, COM+ verifies in every call that the calling identity is a member of at least one of the roles defined for the application, and denies access if it is not. If the caller is a member of at least one role, but the target component does not grant access to any of the roles the caller is a member of, the call is denied access downstream at the component level. Application-level authorization is also the COM+ way of enforcing launch control. The caller cannot launch a new process (by trying to create an object) if it is not a member of at least one role.

7.4.3 Setting the Security Level

The Security Level properties group (which consists of two radio buttons; see Figure 7-10) is the center of the Security tab. This group is the role-based security master switch for all the components in this application. If you set it to the upper position ("Perform access checks only at the process level"), all role-based security configurations at lower levels (component, interface, and method) will be disabled and ignored (see, for example, the bank component security tab in Figure 7-11). When access checks are performed at the process level only, all calls will be allowed through regardless of the settings at the lower levels, as long as they passed the generic application-level security access check.

Figure 7-11. Setting the security access check to be done at the process level only disables component-level security


Bank Component Properties

General | Transactions | Security | Activation | Concurrency | Advanced

Authorization

☑ Enforce component level access checks

Roles explicitly set for selected item(s):

Name
☐ Customer
☐ Teller
☑ Bank Manager
☐ Loan Consultant

One side effect of performing the security checks at the process level only is that you cannot make any programmatic role-based security checks inside your components because the security information will not be part of the call object. You cannot access interfaces such as `ISecurityCallContext`. Additionally, when new objects are activated, COM+ ignores their security requirements when deciding in which context to activate them.
When you set the access security to be performed at the process level and the component level, you can take advantage of role-based security, either administratively or programmatically. COM+ considers the object security requirements when deciding on its activation context. Components that do not want to use role-based security can still choose to do so.
As you can see, disabling component-level security checks globally for an application is of little use to you. You can always disable it on a component-by-component basis.

7.4.4 Setting the Authentication Level

Next, you need to configure the desired authentication level by selecting values from the "Authentication level for calls" combo box (see Figure 7-10). The authentication level controls both caller identity authentication and data integrity authentication. The configured authentication level affects all calls to and from the application.
COM+ lets you set the authentication level to one of six settings: None, Connect, Call, Packet, Packet Integrity, and Packet Privacy. The first four authentication levels deal with the caller's identity only and the last two add data integrity as well.

7.4.4.1 Authentication = None

When the authentication level is set to None, you instruct COM+ not to authenticate the caller at all. If the caller claims to be Joe Customer, then he is believed to be so. Clearly, disabling authentication exposes your application and renders it completely defenseless to anything ranging from innocent user mistakes to malicious third-party attacks. Setting authentication to None may be useful in isolated cases when clients calling in are anonymous and no data privacy or integrity guarantee for data in transit is required. However, you should generally avoid disabling authentication completely.

7.4.4.2 Authentication = Connect

When the authentication level is set to Connect, COM+ authenticates the user identity only when a client connects to an object in the application. Connecting to the object means creating the object or trying to access an object (given to the client from another client) for the first time. COM+ uses the challenge/response protocol to authenticate the client's identity. If the same client tries to connect to another object, COM+ authenticates the client's identity again. However, COM+ stays out of the way once a connection is established. This approach to authentication leaves the door open for a malicious third party to sniff the network, wait for COM+ to authenticate a genuine caller, and then make subsequent calls in place of the legitimate caller, because future calls are not authenticated. Connection-level authentication is the bare minimum required for meaningful role-based security because it verifies at least once that the caller is who it says it is. Connection-level authentication, however, provides no privacy or integrity guarantee for the data in transit.

7.4.4.3 Authentication = Call

When the authentication level is set to Call, COM+ authenticates the caller's identity using challenge/response on every method call to every object in the application, not just the first call. This approach is clearly an improvement over authentication done only at connection time.

7.4.4.4 Authentication = Packet

Authenticating at the beginning of every call may not be secure enough if the method invocation payload is spread over multiple network packets. The underlying network transport protocol may divide the payload (parameters, returned value, source and destination, and so on) over multiple packets regularly. A determined malicious third party may wait for the first packet to be authenticated, and then intercept the rest of the packets, change them, or send his own. To handle this possibility, you can instruct COM+ to authenticate each packet from the caller, not just the first packet of every call. This level of authentication is the default used for every new COM+ server application. Packet level authentication may be the first meaningful authentication setting. However, it still provides no privacy or integrity guarantee for the data in transit.

7.4.4.5 Authentication = Packet Integrity

The previous four authentication levels dealt with authenticating the caller's identity only. Authenticating every packet from the caller would prevent a malicious third party from being tempting to be the caller or pretending to change the packet flow. However, nothing stops a malicious third party from modifying the packets' content. The malicious third party could still, for example, change parameter values inside individual packets.
By setting the authentication level to Packet Integrity, you instruct COM+ to append a hashed checksum to each packet. The receiving side calculates the checksum on the packet just received, and if the resulting checksum differs from that appended to the packet, COM+ fails the call. Packet integrity increases the packet size and network transport time, but it provides a data integrity guarantee. Authenticating data integrity is done on top of packet-level identity authentication.

7.4.4.6 Authentication = Packet Privacy

Although the Packet Integrity level of authentication protects the data integrity of each packet, the malicious third party can still read the packets' content. If you want to protect the privacy of the information, you can instruct COM+ to not only provide packet integrity with a checksum, but also to encrypt the packet's content when in transit and decrypt it when it is received. Packet Privacy is the highest authentication level possible, providing you with authenticated caller identity, data integrity, and privacy for data in transit on every network packet. You will encounter a performance hit for the extra computational effort of encrypting and decrypting every packet. However, for many enterprise applications, this level of security may be required to protect sensitive data properly according to organizational security policy.

7.4.4.7 Deciding on the authentication level

Every authentication setting offers a clear tradeoff of application security versus performance. You should decide on the right authentication level based on the nature and sensitivity of the

services your components expose, potential-threats analysis, and the calling pattern from your clients (the lower the call frequency and the longer the method execution time is, the less noticeable the authentication penalty will be). The application authentication setting affects all components in your application. If the components in your application differ greatly in their authentication needs, consider putting the more sensitive components in a separate application and configuring that application to have a higher level of authentication. Don't make components pay for an authentication level they do not require.
On the other hand, if your threats analysis demands an authentication level that degrades the application performance significantly, or if trade-off is impossible because of organizational security policy, upgrading hardware to improve application performance is an option.

7.4.4.8 Client authentication level compatibility

COM+ prefers to secure the server as much as possible. If the calling client uses an authentication level lower than that of the server (for example, if the client is configured to use Connect and the server application is configured to use Packet), then COM+ fails the call. If, on the other hand, the server is the one using the lower setting, COM+ promotes the connection to the client level.

7.4.5 Setting the Impersonation Level

When an object in Application A calls another object in Application B, identity issues are straightforward: each application has its own identity, used to decide whether to grant access to objects or to resources such as files. However, suppose that Application B needs to access an object in Application C to continue its work on behalf of the original caller in Application A. The immediate question is, under what identity should B access C? Should it access C as B or as A? Suppose that the object in C needs to call back into Application A to complete its work. Should it access Application A as C, B, or A?
One approach would let the server objects impersonate the client. This would be fine in an ideal world, where servers are never malicious. However, in an ideal world, you don't need security either. Clearly, client applications need to declare what identity the servicing objects could use when accessing another application or a secured resource. This is what *impersonation* is all about. The Impersonation level combo box (see Figure 7-10) is at the bottom of the server application security tab. The impersonation level selection is used only when the application you configure is acting as a client of an object in another application. The impersonation level is really a measure of trust—how much this application trusts another application when it acts on its behalf. Does this application

allow other objects to find its security identity? Does it allow them to impersonate itself and perform their work under the client identity, trusting the other applications' objects not to abuse the trust? Does it allow the objects to make additional calls with the original client security identity? These are important questions from any client application perspective. COM+ defines four levels of trust, or impersonation levels: Anonymous, Identify, Impersonate, and Delegate.

> Impersonation of any level requires authentication to be at least Connect (that is, any authentication level except None) to propagate the client identity to the server side.

7.4.5.1 Impersonation = Anonymous

Anonymous is the least trusting impersonation level. The client does not even allow any server object to learn the security identity of the client.

7.4.5.2 Impersonation = Identify

When the client sets the impersonation level to Identify, the server can identify the client—that is, obtain the security identity of the calling client. The server object is not allowed to impersonate the client—everything the object does is still done under the server's own identity. Note that allowing or preventing the object from identifying the caller is not the same as having the object learn programmatically whether the caller is a member of a particular role. When the object queries for the caller's role membership (you will see how later on), the question and the answer are in role terms (Bank Manager, Teller) and not in identity terms (Joe Customer).

7.4.5.3 Impersonation = Impersonate

When the client application sets the impersonation level to Impersonate, the object can impersonate and assume the client identity's credentials. This impersonation level is the default value COM+ uses for new applications. Impersonation indicates a great deal of trust between the client and the servicing object; the server can do anything the client can do, even if the server application is configured to use a less privileged identity. The only difference between the real client and the object is that if the object is on a separate machine from the client, it cannot access resources or objects on other machines as the client. This lack of access is a direct result of the underlying authentication mechanism—the challenge/response protocol. If the object, impersonating the client, tried to access another machine while claiming to be the client, it would fail to authenticate itself as the client because it does not know the client's password. If the object and the client were on the same machine, the object impersonating the client could make one network hop to another machine, since the machine it resides on could still authenticate the client identity—but it could go no further.

7.4.5.4 Impersonation = Delegate

The only difference between delegation and impersonation is that with delegation, the object can freely access any object on any machine as the client. If any of these server objects use delegation, the client identity could be propagated further and further down the call chain. Delegation is possible because Windows 2000 can use the Kerberos authentication service, which uses a different authentication method than challenge/response. Both the client and server user accounts must be configured in the Active Directory properly to support delegation,[1] (in addition to the client granting authority to do delegate-level impersonation), due to the enormous trust (and hence, security risk) involved. Delegation uses, by default, another security service called *cloaking,* which propagates the caller identity along the call chain. Delegation is extremely dangerous from the client perspective because the client has no control over who uses its identity or where. When the impersonation level is set to Impersonate, the client takes a calculated risk because it knows which objects it was accessing. If those objects are on a difference machine, the client identity could not have propagated across the network.

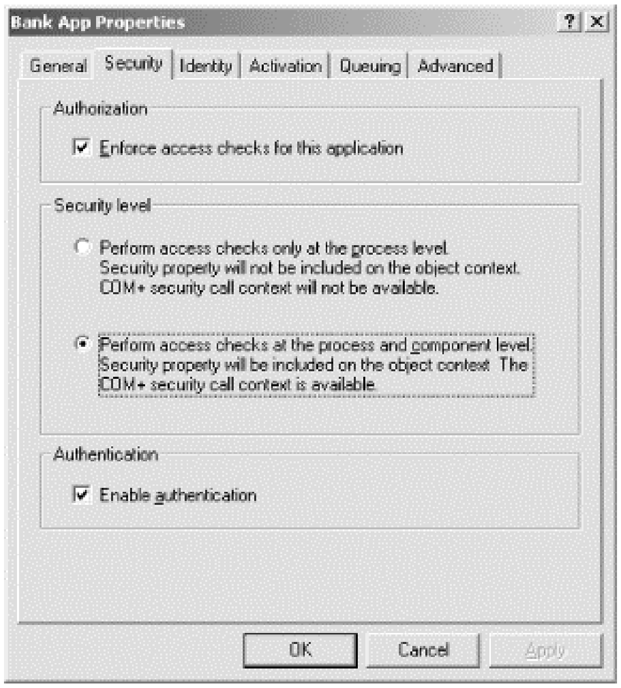
[1] For more information, see *Windows 2000 Administration in a Nutshell* by Mitch Tulloch (O'Reilly, 2000).

7.5 Securing a Library Application

A library application is hosted in its client process. As such, it has no control over the hosting application identity and security settings. It runs under the identity of the hosting process (the Identity tab is still present in the application's properties page, but it is grayed out and ignored). Thus, the library application has only as much privilege as the hosting client does. This limitation may be significant because the library could be loaded by many different clients and may not always have sufficient credentials to do its work. As a rule of thumb, put your meaningful business logic processing components in a server application, where you can configure exactly the application security identity. Deploy a library application in situations when you expect very a intensive calling

pattern from your clients and when you can filter or process the calls before forwarding them to the server application, where the real work should take place. Another identity-related limitation is that a library application cannot declare an impersonation level, so it normally uses the process-wide impersonation level. The library application can set a desired authentication and impersonation level programmatically, as described in Section 7.8 later in the chapter. A library application has no control over the process-level security settings, and the only way for it to perform its own security access checks is to employ component-level role-based security (role-based security at the component level is the same as with a server application). Before you dive into the details of securing a library application, consider the following point: because the library application is loaded into the client process, it has access to all the process resources, memory, objects, GIT, handles, etc. The client should be very careful when loading a library application, as it may contain malicious objects. Agreeing to use a library application implies that the client has a level of trust and familiarity of the library application.
Once you set an application to be a library application, the application's Security tab will be different from that of a server application (see Figure 7-12).

Figure 7-12. A library application's Security tab



Authorization

[✓] Enforce access checks for this application

Security level

( ) Perform access checks only at the process level. Security property will not be included on the object context. COM+ security call context will not be available.

(●) Perform access checks at the process and component level. Security property will be included on the object context. The COM+ security call context is available.

Authentication

[✓] Enable authentication

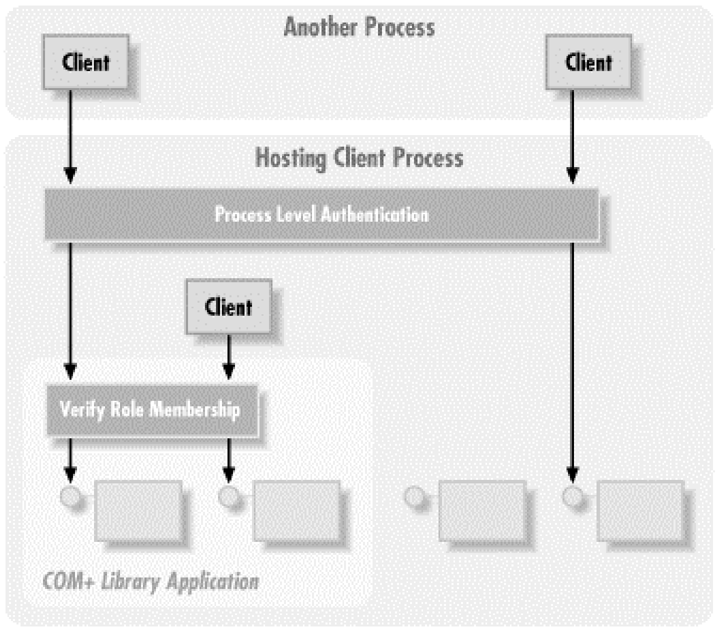
Noticeable by their absence are the authentication and impersonation levels controls, replaced with a single "Enable authentication" checkbox. The authorization checkbox and the security-level radio buttons offer the same functionality as with a

server application. If you want to enable role-based security, the authorization checkbox must be checked and the security level radio button must be at the lower position. This position instructs COM+ to perform access checks at the component level.
The interesting item on this tab is the "Enable authentication" checkbox. The client process hosting this library application can have an authentication level already configured for it. The library application can take advantage of the process-wide authentication and have COM+ use it to authenticate calls coming from outside the process to the library application. However, the library application has no control over how rigorous that authentication is. The process-level authentication may even be set to None. The immediate conclusion is that in a library application, you should avoid performing sensitive work that requires authentication. Therefore, you have at your disposal two mechanisms to secure your library application: process-wide authentication and component-level role-based access control, and you can turn each on or off independently of the other. These mechanisms give you four configuration options, discussed in the following sections.

7.5.1 Both Role-Based Security and Global Authentication

Your typical security setting for a COM+ library application has both role-based security and global authentication enabled. All calls from outside the process are authenticated, whether they are destined for the library application or some other COM object in the process (see Figure 7-13). In addition, COM+ uses component-level access security and verifies that the caller is a member of a role that was granted access to the component. However, calls from within the hosting process are not authenticated. If the hosting process claims to run under the identity of Joe Customer, and Joe is a member of a role that was granted access to a component, clients in the hosting application can access objects in the library application freely. This access opens the way for a malicious client process to load the library application and call into it unauthenticated. This security gap is present in the other three configuration settings as well. This lack of security is yet another reason to avoid performing sensitive work that requires authentication in a library application.

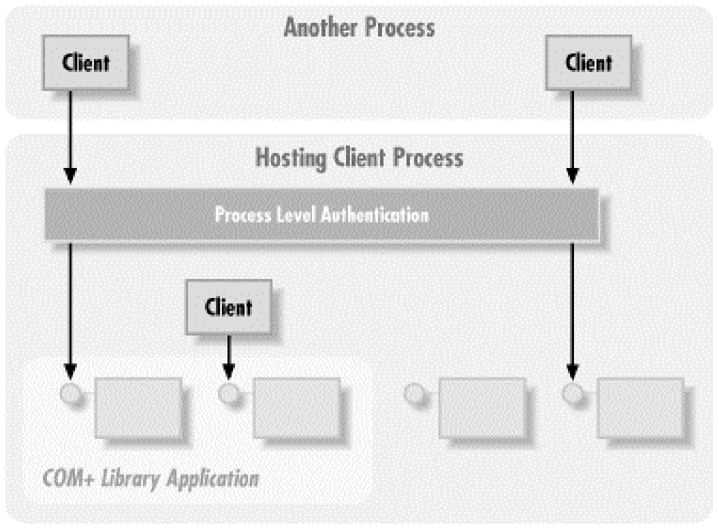
Figure 7-13. Enabling process-level authentication and role-based security



7.5.2 Global Authentication Without Role-Based Security

When importing an existing set of legacy COM components to a COM+ library application (perhaps to be integrated in a bigger development, deployment, and administration framework), the imported legacy components do not use role-based security, and enforcing it may introduce side effects, because those components may already have their own access control mechanisms. It this case, you can turn off role-based security for the library application. As a result, client calls from outside and inside the process access the components directly. However, you still may want to take advantage of the global authentication that may be in place, to authenticate callers from outside the process (see Figure 7-14).

Figure 7-14. Disabling role-based security while relying on global authentication

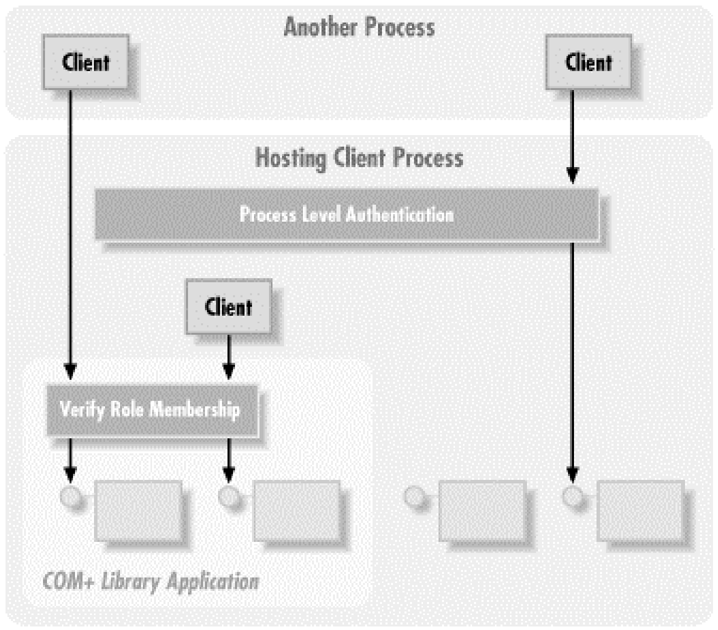


Since you can turn off role-based security at the component level as well, I recommend not disabling role-based security at the library application level. In the future, you may want to add components to the library application that do require role-based security. As a rule, always enable security at the highest level possible, and disable security at the lowest level possible.

7.5.3 Role-Based Security Without Global Authentication

Suppose your hosting process uses a strict authentication level, or at least one that is stricter than what your library application needs. Your application ends up paying a performance hit for a service it does not require. You can choose to disable global authentication support for your application and exempt all calls from outside the hosting process to your library application (see Figure 7-15). However, you should still use role-based security to control access to your application. Of course, there is a downside to disabling authentication: you cannot tell if the callers are who they say they are. You can only decide whether to grant the caller access, assuming the callers are indeed who they say they are.

Figure 7-15. Disabling authentication for the library application while using role-based security

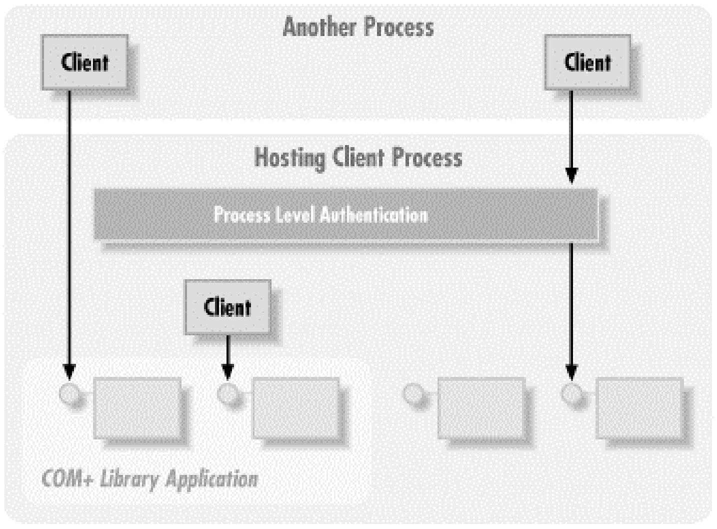


This configuration is usually of little use, as the main motivation for configuring an application as a library is to avoid frequent cross-process calls from clients. If the volume of calls from outside the process is an issue, then just configure the application as a server application, and have your own process-wide authentication level. This configuration has another serious problem: it has the potential for a security breach. Since calls into the library application are not authenticated, what happens if a component in the library application, while executing a method on behalf of an out-of-process caller, tries to access an object in the hosting process (maybe to fire an event on)? Intraprocess calls are not authenticated because all objects in the process share the same identity. Thus, the outside call can bypass the process-wide authentication. This bypass only strengthens the idea that when hosting a library application, the client process should be on guard, should load only library applications it knows are benign, and should minimize their interaction with other objects in the process. At the very least, the hosting server-application should use role-based security, since crossing application boundaries forces access checks and the call from the library application is made across an application boundary. It will not get you authentication, but it will give you some access control.

7.5.4 Neither Role-Based Security nor Authentication

Surprisingly, disabling both role-based security and process-level authentication can be useful. Imagine a situation in which components from your library application are hosted by a browser and have to accept calls from anonymous, unauthenticated callers. The process-wide authentication has to be disabled to allow callers that cannot be authenticated to go through; role-based security cannot be used because you cannot add the anonymous callers to your roles. By turning the security knob all the way down, all calls into your library application will always be granted access (see Figure 7-16).

Figure 7-16. Turning off participation in process-wide authentication and role-based security



7.6 Programmatic Role-Based Security

Sometimes, administrative role-based security it not granular enough for the task at hand. Consider a situation in which your application maintains a private resource (such as a database) that does not expose any public interfaces directly to the clients. You still want to allow only some callers of a method to access the resource and deny access to other callers who are not members of a specific role. The second (and more common) situation is when a method is invoked on your object and you want to know whether the caller is a member of a particular role so you can better handle the call.
To illustrate the second situation, suppose in the bank example, one of the requirements is that a customer can transfer money only if the sum involved is less than $5,000, whereas managers and tellers can transfer any amount. Declarative role-based security goes down only to the method level (not the parameter level) and can only assure you that the caller is a member of at least one of the roles you have granted access to.
To implement the requirement, you must find out the caller's role programmatically. Fortunately, COM+ makes it easy to do just that. Remember that every method call is represented by a COM+ call

object (discussed in Chapter 2). The call object implements an interface called `ISecurityCallContext`, obtained by calling `CoGetCallContext( )`. `ISecurityCallContext` provides a method called `IsCallerInRole( )`, which lets you verify the caller's role membership. `IsCallerInRole( )`, is available on `IObjectContext`, a legacy from MTS as well. Example 7-1 shows how to implement the new requirement using the call object security interface.

Example 7-1. Verifying the caller membership by calling ISecurityCallContext::IsCallerInRole( )

```
STDMETHODIMP CBank::TransferMoney(int nSum,DWORD
dwAccountSrc,DWORD dwAccountDest)
{
   HRESULT hres = S_OK;
   ISecurityCallContext* pSecurityCallContext = NULL;
   _bstr_t bstrRole = "Customer" ;
   VARIANT_BOOL bInRole = FALSE;

   hres = ::CoGetCallContext(IID_ISecurityCallContext,

(void**)&pSecurityCallContext);
   if(pSecurityCallContext == NULL)
   {
      //No security call context available, role-based
security not in use
      return E_FAIL;
   }
   hres = pSecurityCallContext-
>IsCallerInRole(bstrRole,&bInRole);
   pSecurityCallContext->Release( );
   if(bInRole)//The caller is a customer
   {
      if(nSum > 5000)
         return E_FAIL;
   }
   return
DoTransfer(nSum,dwAccountSrc,dwAccountDest);//Helper
method
}
```

7.7 Security Boundaries

COM+ makes a sensible assumption: two components from the same application trust each other, and intra-application security is not necessary. As a result, security is checked only at application boundaries. When two applications interact, a security check exists between them. For example, in the case of a library application that was loaded into a server application, there is an application boundary, and thus a security boundary, between them. When a client accesses the library application in the hosting process, COM+ verifies that the client has access to the library application component. When a client from the library application calls a component in the hosting process, COM+ uses the hosting application's role-based security. The same is true when two library applications interact with each other while both share the same hosting process. You can draw a design conclusion from this behavior: if you have two components and you want security checks done when one calls the other, put them each in separate COM+ applications.

As you have seen, each COM+ method invocation has a call context object associated with it. COM+ will not update the security call context when no security boundary is crossed. If one component has done programmatic role-based security and is about to call another component in the same application, repeating the role membership verification is redundant, as no new security context information will be present.

More on ISecurityCallContext

For most practical purposes, finding out whether the caller is a member of a role is the only part of COM+ security you will ever deal with programmatically. However, `ISecurityCallContext` provides you other extensive security information details, including:

- The total number of callers in the chain of calls leading down to this object.
- The minimum authentication level used to authenticate callers in the calling chain. Even if the immediate caller into this application was properly authenticated, previous callers could have been subjected to less stringent authentication. This may or may not be an issue in your business logic.
- Information about whether a particular user is a member of a role.
- The direct caller's security identity.
- The original caller's security identity.

7.8 Advanced COM+ Security

On top of incredibly rich, user-friendly administrative support for all your security needs, COM+ provides low-level, advanced programmatic security capabilities. These features cater to complex security needs. However, I have found that there is almost always a good design solution that lets me use COM+ configurable settings without having to resort to advanced, programmatic, low-level security manipulation. In fact, you can probably lead a productive and fulfilling development career using COM+ without using low-level security manipulation at all. If what you have read so far fulfills your requirements, feel free to skip this section and move to the conclusion of this chapter and its account of the ever-present pitfalls of COM+ security. If not, continue reading.

7.8.1 Server-Side Impersonation

Setting the allowed impersonation level is a client-side configuration, in which the client declares the level of trust it has toward the server. Configuring the impersonation level is not an advanced security measure; it is a necessary precaution because you cannot know what the server is up to and whether it intends to impersonate the client. However, server-side impersonation is advanced security.

You should be aware that server-side impersonation is not an extensible or scalable design approach. Each COM+ application should be configured with enough credentials (that is, a security identity) to perform its work, and should not rely on the client's credentials by impersonating it. Impersonation couples the server application to the client's identity and prevents the application from evolving independently. In almost all cases when impersonation is a critical part of the application design, the design is not scalable. Consider, for example, an application in which the database performs its own authentication and authorization of end users to secure access to data in the database. Middle-tier objects have to impersonate the caller to access the database, resulting in a programming model that is tightly coupled to the identity of the callers (bank tellers can only access the accounts they are responsible for). Adding new users is not trivial, and therefore does not scale. A better design decision would be to have the database authenticate just the COM+ applications accessing it and trust the applications to authenticate and authorize the clients securely. Allocating database connections per user is another example of when using impersonation is not scalable. The middle-tier objects have to impersonate the user to get a connection. Consequently, the connections cannot be shared (no connection pooling) and the total number of users the system can handle is drastically reduced. One more impersonation liability is performance—impersonating the client can be much slower than making the call directly under the application identity. If the client does not have enough credentials to access a resource, the call fails downstream, when the

impersonating object tries to access the resource, instead of upstream, when the client first accesses the object. Impersonation may also involve intensive under-the-hood traffic and validations.
If you decide to use impersonation, do so judiciously, and only for the purpose of obtaining the client's identity to verify access to a sensitive resource the server application has independent access to. Once the server has verified that the client has enough credentials, the server object should revert to its own identity and access the resource.
The call context object supports another interface called `IServerSecurity`. The server can access `IServerSecurity` by calling `CoGetCallContext( )`. Remember that the pointer to `IServerSecurity` will only be valid for the duration of the current call and cannot be cached.
To impersonate the calling client, the server should call `IServerSecurity::ImpersonateClient( )`. To revert back to its original identity, the server should call `IServerSecurity::RevertToSelf( )`.
Example 7-2 shows a server object impersonating a client to verify that the client has access rights to create a file. If it does, the server reverts to its original identity and creates the file under its own identity.

Example 7-2. The server impersonating the client to verify file creation access rights

```
STDMETHODIMP CMyServer::CreateFile(BSTR bstrFileName)
{
   HRESULT hres = S_OK;

   IServerSecurity* pServerSecurity = NULL;
   hres =
::CoGetCallContext(IID_IServerSecurity,(void**)&pServerSe
curity);
   ASSERT(pServerSecurity);

   hres = pServerSecurity->ImpersonateClient(  );
   HANDLE  hFile =
::CreateFile(_bstr_t(bstrFileName),STANDARD_RIGHTS_ALL,0,
NULL,

CREATE_ALWAYS,FILE_ATTRIBUTE_NORMAL,NULL);

   //Do the cleanup first, before the error handling
   ::CloseHandle(hFile);//Does not change the value of
hFile
   hres = pServerSecurity->RevertToSelf(  );
   pServerSecurity->Release(  );
   if(hFile == INVALID_HANDLE_VALUE)//failure due to lack
of access rights
                                    //as well as anything
else that can go wrong
   {
      return E_FAIL;
   }
   //The client has the right access rights to this file,
now create it again
   //under the server's own identity

   //m_hFile is a member of this object
   m_hFile =
::CreateFile(_bstr_t(bstrFileName),STANDARD_RIGHTS_ALL,0,
NULL,

CREATE_ALWAYS,FILE_ATTRIBUTE_NORMAL,NULL);

   if(m_hFile == INVALID_HANDLE_VALUE)//Something went
wrong
   {
      return E_FAIL;
   }
   return hres;
}
```

COM+ provides two helper functions to automate coding sequences like the one in Example 7-2. `CoImpersonateClient( )` creates the server security object, impersonates the client, and releases the server security object. `CoRevertToSelf( )` similarly creates the server security object, reverts to the server's original identity, and releases the server security object. Example 7-3 shows the same sequence as in Example 7-2, using the helper functions.

> Even though the code in Example 7-3 is more concise and readable than Example 7-2, you should be aware of a slight performance penalty that using the impersonation helper functions introduces. In Example 7-2, the server security object is only created and released once, while it is done twice in Example 7-3. Nevertheless, I recommend using the helper functions because that penalty is truly miniscule and readable code is always essential.

Example 7-3. Verifying file creation access rights with CoImpersonateClient( ) and CoRevertToSelf( )

```
STDMETHODIMP CMyObj::CreateFile(BSTR bstrFileName)
{
   HRESULT hres = S_OK;
   hres = ::CoImpersonateClient( );

   HANDLE  hFile =
::CreateFile(_bstr_t(bstrFileName),STANDARD_RIGHTS_ALL,0,
NULL,

CREATE_ALWAYS,FILE_ATTRIBUTE_NORMAL,NULL);

   ::CloseHandle(hFile);//Does not change the value of
hFile
   hres = ::CoRevertToSelf( );

   if(hFile == INVALID_HANDLE_VALUE)//failure due to lack
of access rights as well
                                    //as anything else
that can go wrong
   {
      return E_FAIL;
   }
   //The client has the right access rights to this file,
now create it again
   //under server own identity

   //m_hFile is a member of this object
   m_hFile =
::CreateFile(_bstr_t(bstrFileName),STANDARD_RIGHTS_ALL,0,
NULL,

CREATE_ALWAYS,FILE_ATTRIBUTE_NORMAL,NULL);

   if(m_hFile == INVALID_HANDLE_VALUE)//Something went
wrong
   {
      return E_FAIL;
   }

   return hres;

}
```

7.8.2 Programmatic Client-Side Security

Every client-side proxy supports an interface called `IClientSecurity`, which lets the client set the security attributes on the communication channel between it and the object behind that proxy. COM+ calls the set of security attributes (such as authentication and impersonation levels) a *security blanket.* Using the `IClientSecurity` method `SetBlanket( )`, the client can set a new authentication level, a new impersonation level, or both. It can

also set other security attributes. However, the proxy may be shared among a few clients, and not all of them may be interested in a new security blanket. COM+ allows a client to clone a personal copy of the proxy for its own use, using another method of IClientSecurity called `CopyProxy( )` that gives the client a private new proxy. You can set a security blanket without cloning your own proxy, but it is recommended that you clone it.
Setting a security blanket may be useful in a few situations:

- When the global application security level is not granular enough. For example, some methods may require additional authentication. In the bank example, the client may want to set an explicit high authentication level for the `TransferMoney( )` method but use whatever security level the application uses for `GetBalance( )`.
- When a library application is at the mercy of the hosting process. If the library application is a client of other objects, though, it can set its own authentication and impersonation levels using `IClientSecurity`.

COM+ provides helper functions to automate coping a proxy (the `CoCopyProxy( )` function) and setting a security blanket (the `CoSetProxyBlanket( )` function). Example 7-4 shows a client of the bank application, copies the proxy, and sets explicit impersonation and authentication levels for the `TransferMoney( )` method, using the helper functions.

Example 7-4. Setting explicit authentication and impersonation levels for the TransferMoney( ) method

```
HRESULT hres = S_OK;
//pAccountsManager is initialized somewhere else
IAccountsManager* pPrivateAccountsManager = NULL;//This
client private copy

//copying the proxy to get a private copy, so not to
interfere with other clients
hres =
::CoCopyProxy(pAccountsManager,(IUnknown**)&pPrivateAccou
ntsManager);

//Setting explicit authentication and impersonation
levels
hres = ::CoSetProxyBlanket(pPrivateAccountsManager,//The
private proxy
                            RPC_C_AUTHN_DEFAULT,//The
system default authentication
                            RPC_C_AUTHZ_DEFAULT ,//Default
authorization
```

```
                           NULL,
                           //Use authentication level
"Packet Integrity"

RPC_C_AUTHN_LEVEL_PKT_INTEGRITY,
                           //Impersonation level is
"Identify"
                           RPC_C_IMP_LEVEL_IDENTIFY,
                           NULL,//Use process identity
                           EOAC_DEFAULT);//Default
capabilities
hres = pPrivateAccountsManager-
>TransferMoney(1000,1234,5678);
pPrivateAccountsManager->Release(  );//Release private
copy
```

I would advise that you always prefer automatic, administrative, declarative security rather than doing security within components, whether it is on the server or the client side. Following this simple rule will make it easier to:

- Write and maintain components
- Design consistent security across an entire application
- Modify an application's security policy without recoding it

7.9 COM+ Security Pitfalls

Distributed systems security is a vast, intricate topic, and certainly COM+ makes it possible for mere mortals to secure systems in an elegant, productive, and extensible manner. All you have to do is understand a few simple security concepts, configure your applications properly, and let COM+ take care of the rest. However, no service is without a flaw, and COM+ security is no exception. Even though the following list of pitfalls may seem long, you should consider two things: first, considering how encompassing COM+ security really is, it is a surprisingly small list, as security affects almost everything you do in COM+. Second, this list describes only things I have encountered, and it is probably only partial. You will undoubtedly encounter other variations and pitfalls when you do your own development. However, with a solid understanding of the way COM+ security works, you should be able to isolate and troubleshoot the problems yourself. Some of the pitfalls have already been implied throughout this chapter, but the following is dedicated and explicit pitfall list.

7.9.1 Machine-Wide Security Settings

At the root of the Component Services Explorer is the My Computer item, which lets you set global configurations for your computer. If you have administrative privileges on other machines, you can add them to the list of machines managed by the Component Services Explorer. Each computer icon has a properties page with two tabs that are seemingly relevant to COM+ security: the Default Security tab and the Default Properties tab.

Though these tabs are part of the Component Services Explorer, they have little or nothing to do with COM+ applications and are the reincarnation of `DCOMCNFG.EXE`, the awkward DCOM configuration utility. The Default Security tab has no bearing on COM+ applications. It is used to control default access and launch permission for classic COM local servers. The Default Properties tab is mostly irrelevant for COM+ applications. It is used to set default authentication and impersonation levels for COM local server processes that can be accessed remotely. If those processes were to interact with COM+ applications as clients to the configured objects (locally or remotely) and did not provide their own security configurations (administratively or programmatically), then these settings would be used. It short, neither tab is relevant to COM+ applications.

7.9.2 Calling CoInitializeSecurity( )

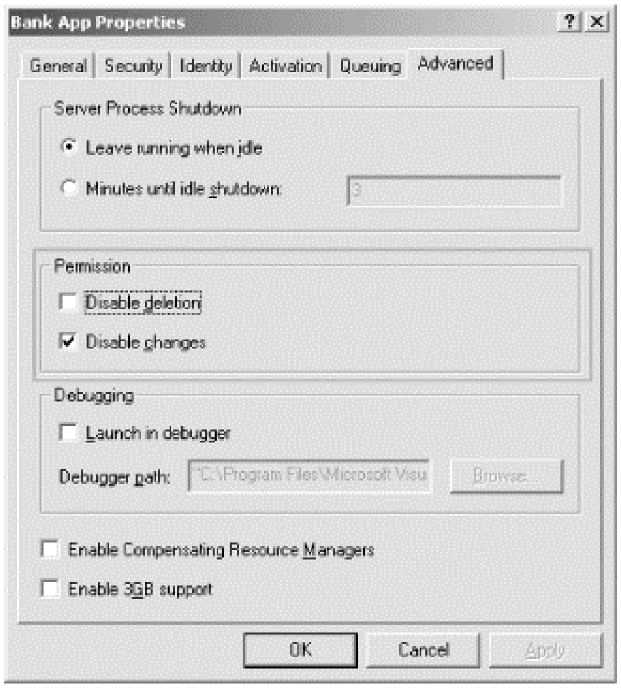
If you used DCOM security before, calling `CoInitializeSecurity( )` is second nature to you. In the old DCOM days, `CoInitializeSecurity( )` was the gateway to manageable security, and any properly written DCOM server called it to ensure that the required security levels would be used. However, a configured component has no point in calling `CoInitializeSecurity( )` because any configured component is loaded in a hosting process. If the component is part of a server application, COM+ calls `CoInitializeSecurity( )` when the process is created, with the application global security settings as parameters. If the component is part of a library application, the hosting process calls `CoInitializeSecurity( )` before doing anything else with COM. Otherwise, COM would have called `CoInitializeSecurity( )` for it.

`CoInitializeSecurity( )` may be an issue when importing an existing COM local server to COM+. If the ported server used `CoInitializeSecurity( )`, you must remove the call from the code, look at the parameters for `CoInitializeSecurity( )`, and configure the global application security levels accordingly.

7.9.3 Disabling Changes to the Application Configuration

A permission properties group is located on every COM+ application's Advanced tab (see Figure 7-17). By selecting "Disable changes," you can prevent anybody from making changes to your application settings (and any changes at the component, interface, and method level), including the security settings and access policy. The problem is that this checkbox is not password protected, and anyone with administrative privileges can modify your precious security settings and introduce security gaps in your application. Customer-side administrators (who are not your product administrators) may be tempted to change your security settings to accommodate something else in the system, or just to fool around with your application. Be aware of this situation. This checkbox is there for a reason, and I wonder why Microsoft did not take an extra step and make it password protected.

Figure 7-17. Disabling and enabling changes to your application



Server Process Shutdown
- [✓ Leave running when idle]
- Minutes until idle shutdown: 3

Permission
- ☐ Disable deletion
- ☑ Disable changes

Debugging
- ☐ Launch in debugger
- Debugger path: "C:\Program Files\Microsoft Visu

☐ Enable Compensating Resource Managers

☐ Enable 3GB support

7.9.4 Avoid Sensitive Work at the Object Constructor

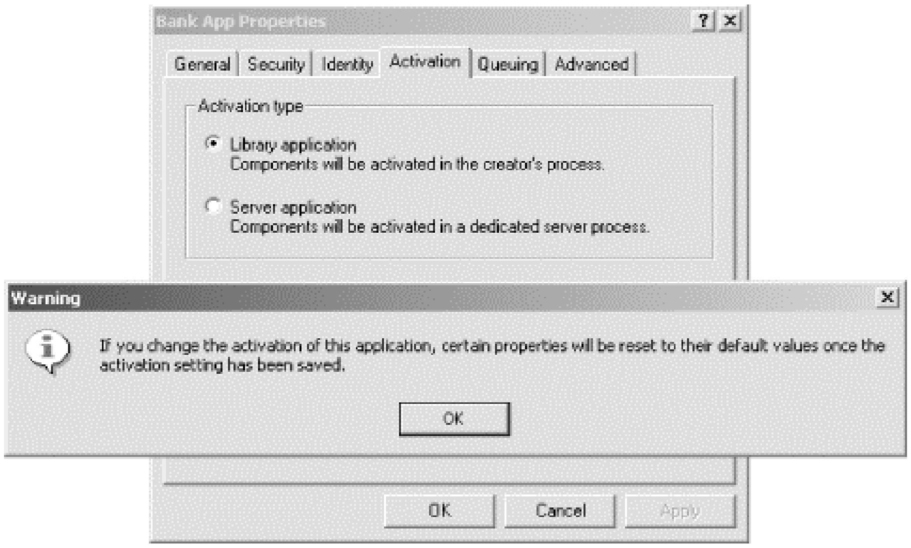
Imagine a situation in which a client is granted access (using role-based security) to one component in your application, Component A, but is not granted access to another component, Component B. When the client tries to create Component B, COM+ creates the object, but only lets the client access the `IUnknown` methods of Component B and denies access to methods on any other interface. As explained in the Launch Control definition at the beginning of this chapter, this process intentionally avoids a DCOM pitfall. This pitfall allows a client to create a new object in a new process, but forgets

to grant the client access to the objects inside. This pitfall resulted in a zombie process because the client could not even call `IUnknown::Release( )` on the object it just created.
However, because the client is allowed to create the object, it implies that the constructor of Component B actually executes code on behalf of a client that is not allowed to access the component. If you do any sensitive work at the object constructor, it may constitute a security breach because that work should never be done for that client. The obvious conclusion is to avoid doing any sensitive work in the object constructor, such as erasing or opening sensitive files or creating sensitive accounts in a database.

7.9.5 Changing Application Activation Mode

When you switch between application activation modes (for example, from a library to a server application), COM+ presents you with the enigmatic message box shown in Figure 7-18. The message box warns you that certain properties will be set to their default values. Those settings are mostly security properties that the library application does not have, such as authentication and impersonation settings. After changing the application activation mode, go through the security settings and make sure the default values COM+ picked up for you are what your design calls for, and set them to the correct values if you need to.

Figure 7-18. COM+ warns you that some settings have been set to their default values



(●) Library application
Components will be activated in the creator's process.

( ) Server application
Components will be activated in a dedicated server process.

If you change the activation of this application, certain properties will be reset to their default values once the activation setting has been saved.

7.9.6 IsCallerInRole( ) Returns TRUE When Security Is Not Enabled

Programmatic role-based security, as you have seen, is used to verify the caller's membership in a particular role. However, role-based security must be enabled properly for `ISecurityCallContext::IsCallerInRole( )` to return accurate results. In the following cases, `IsCallerInRole( )` always returns `TRUE`, regardless of the actual caller role membership:

- Role-based security is enabled at the application level, but not enforced at the component level, because the "Enforce component level access checks" checkbox (shown in Figure 7-3) is not selected. Calls to `ISecurityCallContext::IsCallerInRole( )` from within the component always return `TRUE`.
- At the application level, authorization is not enforced because the "Enforce access checks for this application" checkbox (shown in Figure 7-10) is not checked. All calls to `ISecurityCallContext::IsCallerInRole( )` will always return `TRUE`, even if component level access checks are enabled.

`IsCallerInRole( )` misbehaves in both library and server applications when one of these two situations occurs.
To overcome this misbehavior, you should call another method of `ISecurityCallContext` to verify that security is enabled before checking role membership. This method is called `IsSecurityEnabled( )`, and is available specifically for these cases. Example 7-5 shows the same code as Example 7-1, except this time `IsSecurityEnabled( )` is used first.

Example 7-5. Verifying that security is enabled before checking the caller role membership

```
STDMETHODIMP CBank::TransferMoney(int nSum,DWORD
dwAccountSrc,DWORD dwAccountDest)
{
   HRESULT hres = S_OK;
   ISecurityCallContext* pSecurityCallContext = NULL;
   _bstr_t bstrRole = "Customer" ;
   VARIANT_BOOL bInRole = FALSE;
   VARIANT_BOOL bSecurityEnabled = FALSE;

   hres = ::CoGetCallContext(IID_ISecurityCallContext,

(void**)&pSecurityCallContext);
   if(pSecurityCallContext == NULL)
   {
      //No security call context available, role-based
security not in use
```

```
      return E_FAIL;
   }
   hres = pSecurityCallContext-
>IsSecurityEnabled(&bSecurityEnabled);
   if(!bSecurityEnabled)
   {
      pSecurityCallContext->Release(  );
      return E_FAIL;
   }
   hres = pSecurityCallContext-
>IsCallerInRole(bstrRole,&bInRole);
   pSecurityCallContext->Release(  );
   if(bInRole)//The caller is a customer
   {
      if(nSum > 5000)
         return E_FAIL;
   }

   return
DoTransfer(nSum,dwAccountSrc,dwAccountDest);//Helper
method
}
```

7.9.7 Disabling Application-Level Authorization

When you disable application-level authorization, even if a component is set to use and enforce role-based security (as in Figure 7-3), all calls to that component will be permitted, regardless of the caller's identity and role membership. This situation is very dangerous, as the component, by design, may require access control and does not have another mechanism in place to implement access control requirements.
In addition, unlike the case of setting the security level to process-wide only (which disables component level role-based security and allows all calls), the component security tab will not be grayed out as in Figure 7-11. Always leave the application-level authorization enabled.

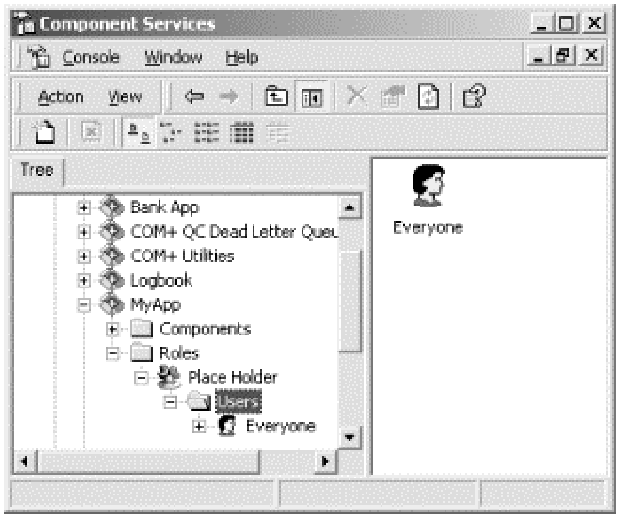
7.9.8 Enabling Application-Level Authorization

As explained in the previous pitfall, you should always enable application-level authorization. However, what happens if, in your application, you have a number of components that require role-based security and a few other components that do not? The components that do not require access control may serve a different set of clients altogether. Application-level authorization is problematic because when a call comes into an application, COM+ verifies that the caller is a member of at least one role defined for this application. If the caller is not a member, COM+ denies the caller access, even if the caller tries to access a component that does not require access control.

There are two ways around this pitfall. The first is to move the components that do not require role-based security to a separate application. The second solution simply defines a new role in your application called Place Holder and adds just one user to it: the Everyone group (see Figure 7-19). Now all callers are members of at least one role, and components that do not require role-based security can accept calls from any user while application-wide authorization is enabled.

Be aware that using the Place Holder role with the Everyone user in it actually moves the first line of defense to the component layer instead of the application layer. This movement may open the way for a denial of service attack by a malicious client that bombards your application with requests to create new components. COM+ allows the attacker to create the components, but not access them. The bombardment may cause your application to run out of resources.

Figure 7-19. Adding a role as a placeholder for the Everyone user



7.10 Summary

COM+ security offers the component developer a wide spectrum of security support, from simple and administrative role-based security to advanced programmatic security. Security is all about tradeoffs: performance versus risk mitigation, ease of use versus flexibility, and ease of administration versus potential attacks. Regardless of where you find yourself in this spectrum, you will learn to appreciate the elegance and power of COM+ security.

You can also combine COM+ security with the high-level COM+ services described in the next chapters: COM+ queued components and COM+ loosely coupled events.

Chapter 8. COM+ Queued Components

COM+ Queued Components is a service that allows a client to call object methods asynchronously. The client is blocked only for a short duration while COM+ processes the request, and the object executes the call at a later point. You can think of queued components as *asynchronous COM+*.

Under classic COM and DCOM, all method calls on your object are synchronous—the client is blocked while the object executes the call. Classic COM developers often had to develop a proprietary mechanism for asynchronously invoking calls on their objects. One recurring mechanism had the object spin off a worker thread to process the client request and immediately return control to the client. The object would later signal the client somehow when the call completed (if the client needed to know), and the client had to distinguish between method completions of multiple objects.

Such solutions coupled the clients to the objects and were inconsistent. Different vendors provided slightly different solutions, requiring different programming models on the client side at times.

The first release of MTS and Microsoft Message Queue (MSMQ) in 1998 provided another way to support asynchronous object method calls with COM. MSMQ is a message queuing service that allows you to post messages from one queue to another, potentially across machines.

Clients and objects could use MSMQ to facilitate COM asynchronous method calls. With MSMQ, the client posts a message to a designated queue that contains the method name and parameters. The object retrieves the message off the queue, parses the message, and executes the call. The object and client developers agree about the queue location, the message format, and other details required for asynchronous interaction in advance.

However, using MSMQ for asynchronous calls has some disadvantages:

- The nonstandard interaction couples the object to its clients.
- The client developers still have to design and implement a way to package the method information into a message, and object developers still have to design and implement a way to parse the message.
- MSMQ is not easy to install and use. Developers have to learn how to write code to use MSMQ interfaces.
- The client is very much aware that it uses MSMQ to post the call to the object. The resulting asynchronous method invocation code does not resemble the synchronous method invocation on the same COM interface.

This approach is analogous to the pre-DCOM days when developers wrote raw RPC calls to invoke methods on remote objects.
The idea behind COM+ queued components is simple: let COM+ encapsulate the interaction with MSMQ and provide a uniform way of invoking asynchronous method calls. In fact, the method invocation code itself is the same as a synchronous call. The only difference is in the way the client creates the object.
You can think of MSMQ as the transport layer between the client and object, much like RPC is the transport layer in the case of remote activation. A DCOM client does not care about the underlying details of RPC protocol and marshaling when invoking a method on a remote machine. Similarly, a queued components client does not need to care about the details of the MSMQ protocol and the methods-to-message conversion.
Queued components are an essential addition to your arsenal because implementing robust asynchronous execution on your own is a demanding task; it requires you to spend much effort on design, implementation, and testing. By providing you with queued components, COM+ lets you focus on the domain problems at hand, rather than on complicated asynchronous plumbing.

8.1 Major Benefits of Queued Components

Besides simplifying asynchronous method invocation, queued components provide you with other major benefits (discussed in the following sections).

8.1.1 Disconnected Work

When the client calls a queued component, the call is converted to a message and placed in a queue. MSMQ detects the message in the queue and dispatches the message to the queued component. If the client and the object are on different machines, the message can be placed in a local queue on the client machine, if necessary.
Imagine that the client is disconnected from the network: suppose a sales person is working on a laptop at the airport while waiting for a flight. The client application on the laptop can still make calls to queued components—to update order numbers, for example. The calls are stored locally by MSMQ. The next time the client machine is connected to the network, MSMQ is aware that the local queue contains messages, so it dispatches them to the remote component.
The server hosting the objects could be disconnected as well. MSMQ transfers queued messages from the client machine once the object machine is brought back online.
The benefits of disconnected work are twofold. First, your system's robustness improves because network outage between a client and

a queued component is handled easily. Second, allowing disconnected work in your application, by design, has practical importance: approximately 40 percent of all new computers sold are for mobile and portable use. These devices benefit greatly from queued components, as they allow users to continue working while offline or on the road. Targeting the portable market is an important consideration for many modern applications.

8.1.2 Real Life Business Model

Many enterprise-wide applications are developed to automate existing business processes and information flow. These processes, such as email and voicemail, are often messaging-based by nature, and modeling them with queued components is very appropriate.

8.1.3 Component Availability

A component may not be available because of server overload or networking problems. Under classic DCOM, you would have to abort the whole transaction or wait for the component to become accessible. Using queued components, you can separate the transaction into activities that must be completed now and those that can be completed later. Your end users will be unaware of server slowdowns or failures.

8.1.4 MSMQ Participates in Transactions

MSMQ is a resource manager, and will thus auto-enlist in your transactions. When your application makes calls to queued components during a transaction, your application (via COM+) adds messages to an MSMQ queue. Those messages will not persist in the queue if the transaction is aborted. The transaction coordinator (DTC) instructs all resource managers that participated in the transaction to roll back the changes. MSMQ's rollback rejects the messages that were added to the queue during the transaction.

8.1.5 Auto-Retry Mechanism

Once a message is added to a queue, COM+ tries to invoke the call in that message on the object. When COM+ retrieves the message from the queue, it creates a new transaction for the retrieval. If the object participates in that transaction, and that transaction is aborted, MSMQ's rollback in this case will return the message to the queue. This, in turn, causes COM+ to try again to invoke the call on the object.

8.1.6 Scalability

A major scalability bottleneck is the length of time the client ties up an instance of the server. In a distributed system, you should minimize that time as much as possible by reducing the number of network round trips to allow your server to accept calls from other clients. When a client makes calls on a queued component, COM+ records the calls the client makes and combines them into a single message. Message delivery generally requires just a single network operation, so the time the server instance is occupied is reduced.

8.1.7 Workload Buffering

Every system has a peak load of clients asking for services. Systems architects have to design the system to handle that peak load. The question is, what do you do if the workload is uneven throughout the day? Designing your system to handle the peak load in real time may require huge investments in expensive hardware, load balancing machines, longer development time, and more difficult design goals. Such an approach results in a system that may handle the peak load, but remains vastly underutilized on average. A more realistic alternative is to accept client requests, buffer them, and execute them later on. For example, most online web stores do exactly that—they accept your order immediately and you are free to surf other web sites. The store buffers your request and can handle the next client. In the background, at the system's leisure, it processes the request and sends you an email confirmation once your order is processed and shipped.

Using queued components, you can separate the purchasing task into two stages: a short-duration, front-end, synchronous acknowledgement, and an offline, queued task—the order processing itself.

8.1.8 When Should You Use Queued Components?

Clearly, queued components offer solutions to several real-life problems, saving you precious development time and increasing overall system quality. The question is, when should you use queued components?

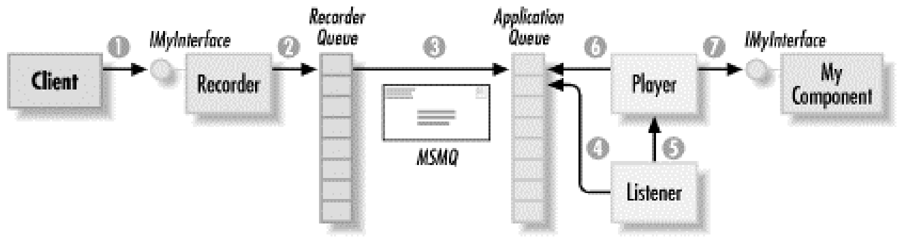
During system requirements analysis, try to identify business activities that can be separated by time. You may execute each activity synchronously, but you connect them with queued components.

For example, imagine an online store. Orders are collected from the customers immediately and synchronously. Processing the order—parts orders to various vendors, billing updates, and so on—can be done later. All tasks must be done, but they don't all have to be done at once.

8.2 Queued Components Architecture

One of the major requirements for the COM+ queued components architecture specifies that the component developer should take no special steps to make a component asynchronous; the developer writes synchronous code, and COM+ provides the mechanism to allow clients to call the method asynchronously.
As a result, the client cannot create the component directly, since it would result in the usual blocking calls. Instead, COM+ uses the architecture shown in Figure 8-1. For every queued component, COM+ provides a *recorder object.* The recorder object supports the same set of interfaces as the queued component. When the client calls methods on the recorder interfaces (Step 1), the recorder (as the name implies) merely records the calls. When the client releases the recorder, the recorder converts the calls to an MSMQ message and posts that message to the recorder queue (Step 2).
Every application that contains queued components has a queue associated with it. MSMQ transfers the message to the application queue from the recorder queue (Step 3). For each application, COM+ maintains a *listener object* that detects when a message was added to the application queue (Step 4). The listener creates a *player object* (Step 5) and instructs it to retrieve the message from the queue (Step 6). The player creates the actual component and plays the calls back to it (Step 7). When the player is finished playing back calls, it releases the component.

Figure 8-1. COM+ queued components architecture



8.2.1 The Recorder

You can think of the recorder as the component proxy. The recorder is responsible for forwarding the call across processes or machines to where the object resides. The recorder lives in the client process and supports the same set of queued interfaces as the component. When clients query the recorder for a different interface, then the recorder must also provide recording ability for the interface if it is supported by the real component.

8.2.2 The Player

The player in this architecture is analogous to the stub—it translates the MSMQ message to method calls and then makes those calls to the object. The player is responsible for removing the message from the queue and is configured to always require a transaction. As a result, creating the player kicks off a new transaction that includes in its scope the message removal and the playback of method calls. Every action the queued component takes when executing the methods, such as database updates, executes within that transaction. If, for example, the database update fails, the transaction aborts and every resource manager that took part in it has to roll back. As mentioned previously, MSMQ is a resource manager and its rollback puts the message back in the queue. The listener detects its presence there and retries the playback sequence (more on that later).

8.2.3 The Listener

Every COM+ application has at most one listener associated with it, serving all queued components in the application by listening to the application queue and creating the player objects.
Note that the queued components design separates the act of detecting a message from the act of playing it back to the component. If the listener were responsible for calling methods on the objects, then all calls to queued components would be asynchronous, but serialized—that is, occurring one at a time. That kind of design would have killed performance. By having a dedicated player for each component, the listener can process asynchronous calls as fast as they can be added to the queue.
The listener object lives in the application process. If you configure your application to support queued components, COM+ creates a listener in the application process when the application is launched. In fact, if the application is not running, then no one will listen to its message queue, and, as a result, no queued components will ever be instantiated. COM+ cannot possibly know when it is a good time to create the application and have it start servicing queued calls for you. Only the application administrator has that knowledge (for example, what hours of the day or what load level).
You have a number of options available for launching the application:

- Start the application manually from the Component Services Explorer.
- Provide your application administrator with a simple utility that makes programmatic calls to the COM+ Catalog (as explained in Chapter 6) to start the application.
- Use the Windows Task Scheduler to invoke your utility at designated times.

- Activate nonqueued component in the same application. This activation causes COM+ to launch the application, and by doing so, it creates the listener.

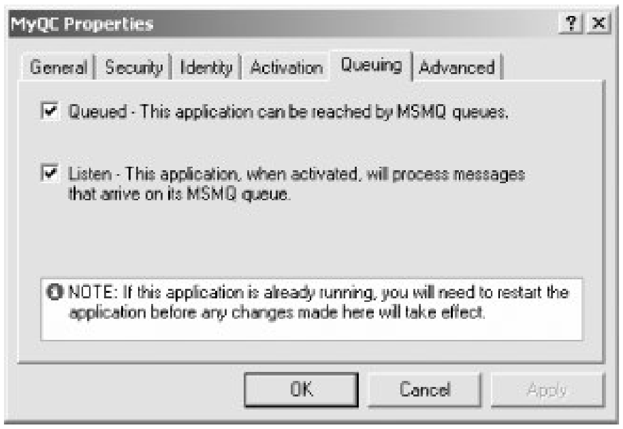
8.3 Component Services Explorer Configuration

Before you begin configuring the Component Services Explorer, make sure you have MSMQ installed on your machine. The Windows 2000 installation does not install MSMQ by default. To add MSMQ to your system, go to the Control Panel and click on Add/Remove Programs. In the dialog box, click Add/Remove Windows Components, and instruct the wizard to install Message Queuing Services. This step starts the MSMQ installation. Choose the Workgroup installation for a single-machine setup, or if you have a domain account on a domain server, choose the domain installation for secure cross-machine invocations.

8.3.1 Application Configuration

Every COM+ Server application can host queued components. On the application properties page, a Queuing tab (see Figure 8-2) enables and configures queued component hosting by that application. The tab contains two checkboxes, "Queued" and "Listen".

Figure 8-2. The COM+ server application Properties page's Queuing tab



☑ Queued - This application can be reached by MSMQ queues.

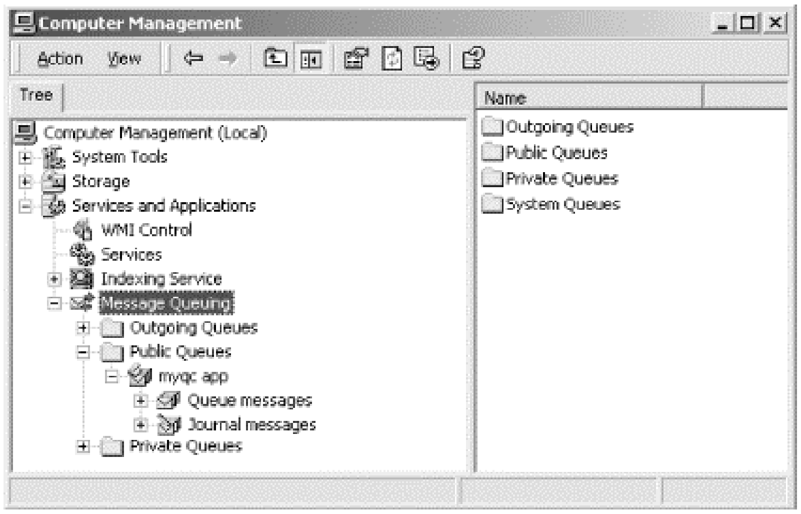
☑ Listen - This application, when activated, will process messages that arrive on its MSMQ queue.

NOTE: If this application is already running, you will need to restart the application before any changes made here will take effect.

Checking the Queued check box causes COM+ to create a public message queue, named as the application, for the use of any queued components in the application. Incoming messages for queued components in the application are posted to that queue. You can actually see the queue associated with your application by using the MSMQ Explorer. To bring up the MSMQ Explorer, go to the Control Panel, open the Administrative Tools folder and expand

Computer Management →Services and Application →Message Queuing. You will see all the MSMQ queues installed on your computer. If, for example, your COM+ application is called MyQC App, once you check the Queued check box, under the Public Queues folder you should see a new queue called *myqc app* (see Figure 8-3).

Figure 8-3. Using the MSMQ Explorer, you can see the queue associated with your application

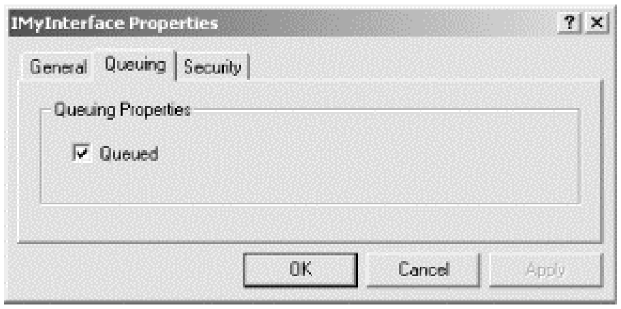


Checking the "Listen" checkbox on the Queuing tab instructs COM+ to activate a listener for the application when the application is launched.
Normally, if you have queued components in the application, you should have the "Listen" checkbox checked. However, COM+ allows you to turn off processing queued calls (by unchecking the "Listen" checkbox) to allow nonqueued components in the application to sever their clients adequately without the performance hit of the queued calls. The performance can be sustained at a later point in time.
A COM+ library application cannot contain COM+ queued components because it is hosted at runtime by a client process, over which COM+ has no control. In fact, the client process may not even be a COM+ server application. COM+ cannot create MSMQ queues as needed for a process or inject listener objects into it. If you use queued components, you must put them in a server application.

8.3.2 Queued Component Interface Configuration

The fact that a client wants to make asynchronous calls on a component does not mean that the component developer allows it. You have to enable queuing for every interface on which you expect to receive asynchronous calls. You do that by displaying the interface properties page and then selecting the Queuing tab. The tab has a single checkbox (see Figure 8-4). When checked, that interface on your component can accept queued calls.

Figure 8-4. The interface Properties page's Queuing tab



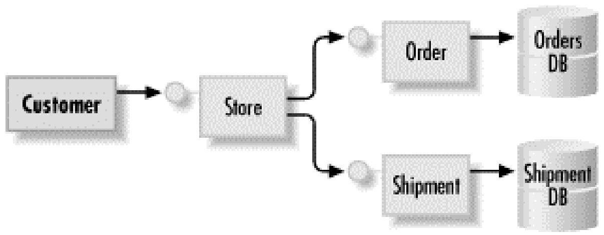
Queuing Properties

☑ Queued

8.4 Invoking Queued Components on the Client Side

A queued component client cannot create the component using `CoCreateInstance( )` (or `CreateObject( )/New` for Visual Basic 6.0) because it would result with the normal synchronous mode of interaction. The client must create the component in a way that would make COM+ create a recorder for the calls instead.
Consider, for example, the system in Figure 8-5, which shows the component layout of an online retail shoe store. The customer interacts with a top-level Store component. The interaction with the customer must be fast and synchronous. The customer specifies shoe size, shipping method, email address, credit card number, and so on. The Store component saves the order information using the Order component and processes the order using the Shipment component.

Figure 8-5. A simple online retail store system containing Store and Order COM+ components and a queued Shipment component



However, shipping the order (ordering the shoes from the vendor, updating inventory, interacting with the shipping company, etc.) can take a long time to complete. Processing and shipping the order should be done offline, using COM+ queued components.
The Shipment component exposes the `IShipment` interface, defined as:

```
interface IShipment: IDispatch
{
   [id(1)] HRESULT ShipOrder([in]DWORD dwOrderNumber);
}
```

The Shipment component prog-ID is `MyApp.Shipment`.

The first step in using queued components configures the application containing the Shipment component to host queued components, and then configures the `IShipment` interface on the Shipment component, as shown in the previous section.

The client of a queued component creates a recorder for the calls made using a *moniker* —a string that shows how to bind to an object. If the client is written using Visual Basic, the client uses the `GetObject( )` call. A C++ client would use the equivalent `CoGetObject( )`.

For example, if the Store component were implemented using Visual Basic, you would write the following code to create a recorder for the queued Shipment object and execute the call:

```
orderNumber  = 123
Dim Shipment As Object
Set Shipment = GetObject("queue:/new: MyApp.Shipment")
Shipment.ShipOrder(orderNumber)
```

And if it were written in C++:

```
IShipment* pShipment = NULL;
HRESULT hres = S_OK;
DWORD dwOrderNumber = 123;

hres = ::CoGetObject(L"queue:/new: MyApp.Shipment", NULL,
                     IID_IShipment,(void**)&pShipment);

hres = pShipment->ShipOrder(dwOrderNumber);

pShipment->Release(  );
```

Alternatively, the C++ client can create the queued component using the component CLSID instead of the nonunique prog-ID:

```
hres = CoGetObject(L"queue:/new:{8B5C3B80-6D0C-49C7-8F74-
14E59D4BEF40}",...,);
```

Nothing in the client's code differs from interacting with the same component synchronously, except creating the object.

COM+ actually uses two monikers to create the queued component. The first is the `new` moniker that has existed since the early days of COM. The new moniker, an alternative to `CreateObject( )` and `CoCreateIntance( )`, is used to create a component.

For example, the following two Visual Basic lines are equivalent:

```
Set Order = CreateObject("MyApp.Shipment")

Set Order = GetObject("new: MyApp.Shipment")
```

Either line would create the nonqueued component.

The `queue` moniker is a new addition, introduced by COM+ to support queued components. The combination of `queue:/new:` tells COM+ to create a recorder instead of the real object. For practical purposes, the syntax used to create a queued component (shown previously) is all you will ever need.
However, COM+ provides the queued component client with many extensions to the queued moniker that allow you to override its default behavior. These extensions include:

- Posting the recorded method calls to a specified queue, instead of the one associated with the queued component application. You can specify the queue name and location (the machine on which the queue resides), as well as application-specific information that will be attached to the message.
- The message security authentication level.
- The message encryption options.
- Whether MSMQ should keep a journal of the message.
- Various send and receive timeouts.
- The message priority within the queue.

Please refer to the MSDN Library for more information about these and other extensions. The very fine-grained control a client can have over the queued method recording is another reason why the conventional mechanism for creating components (`CoCreateInstance` or `New`) cannot be used for queued components.

8.5 Designing Queued Component Interfaces

When a client makes calls to a queued component, it interacts with the recorder provided by COM+. No actual calls to the real object occur. So, at the time of the call, the client has no way to receive output from the method, nor can it tell whether the call succeeded or failed. Consequently, when you design an interface to be supported by a COM+ queued component, you must avoid any outgoing parameters on any interface method.
Specifically, do not use any `[out]`, `[in,out]`, or `[retval]` IDL attributes on your method parameters. When you import a component into the Component Services Explorer, COM+ inspects the interfaces supported by that component, and if it detects an output attribute, COM+ disables the queuing option for that interface.
If you develop your COM+ component using Visual Basic 6.0, you do not have direct access to your component IDL. Normally, this lack of access would not be a problem. However, Visual Basic, by default, treats method parameters as `[in,out]` parameters. If you

expect your component to be accessed as a queued component, you have to explicitly use the Visual Basic `ByVal` attribute on your method parameters.

> In the next version of Visual Basic, Visual Basic.NET, all parameters are, by default, passed in by value instead of by reference. See Chapter 10 for more information.

A different kind of a parameter returned from a method is its return value. You should avoid using custom `HRESULT` codes to indicate particular failure cases. The client only receives the `HRESULT` from the recorder, indicating the success or failure of recording the call. When your object executes, its `HRESULT` codes are delivered to the player, which does not understand your custom semantics. COM+ does not require you to stick to the standard `HRESULT` codes, but there is no point in not doing so.

Another restriction on queued component interfaces is that the client cannot pass in a pointer to another COM object. The reason is obvious—when the actual call takes place later, there is no guarantee that the object passed in as a parameter is still around.

Implementing a queued component implies more than just a few method parameters restrictions and clicked checkboxes on the Component Services Explorer. It really means a change in your design patterns and programming model.

The only way to pass in COM objects as a method parameter to a queued object is if the object you pass in supports the interface `IPersistStream`. `IPersistStream` is a standard interface used for object serialization, defined in the early days of OLE2 and COM. Objects that support `IPersistStream` can serialize their state into a COM+ provided stream and reconstruct their state out of such a stream.

Whenever you pass a COM object as a method parameter, COM+ queries it for `IPersistStream`. If the object supports it, COM+ calls `IPersistStream::Save( )`, passing in a pointer to an `IStream` object. The input object saves its state to the stream. The recorder stores the saved state information in the message sent to the queued component application queue. When the player retrieves the message, it creates a stream object from the message and calls `IPersistStream::Load( )` to initialize the object to the state it was in when the call was made. It then invokes the call on the component, passing in the input object as parameter.

When you design an interface used by a queued component, you can use a new IDL extension, an interface attribute called `QUEUEABLE`, to denote it as an interface used by a queued component. The *mtxattr.h* header file defines this extension.

Example 8-1 shows the IShipment interface definition again, this time using the QUEUEABLE attribute.

Example 8-1. Using the IDL QUEUEABLE attribute to denote an interface as queued-components compatible

```
#include "mtxattr.h" // For QUEUEABLE
[
   object,
   uuid(97184D0F-F7EF-413A-8C6D-C1745018B2E9),
   dual,
   pointer_default(unique),
   QUEUEABLE
]
interface IShipment: IDispatch
{
   [id(1)] HRESULT ShipOrder([in]DWORD dwOrderNumber);
};
```

This attribute autoconfigures the interface as Queued when you import a component that supports the interface into the Component Services Explorer. This autoconfiguration saves you the trouble of doing it yourself. The attribute has no semantic meaning for the MIDL compiler; it will not stop you from defining [out] parameters on an interface marked QUEUEABLE. Only COM+ examines this attribute.

ATL 7 Queuing Attribute

As explained in Chapter 4, if you use the attributed ATL 7 project under Visual Studio.NET, you can take advantage of precompiler-specific support for COM+ 1.0 services, using special attributes. If you add a new class to your ATL 7 project, and you select "ATL COM+ 1.0 Component" from the Add Class dialog, the wizard will let you configure queued-component support for the interface. If you select the "Queueable" checkbox, the attributed interface will have a custom attribute as part of its deceleration:

```
[
   object,
   uuid(97184D0F-F7EF-413A-8C6D-C1745018B2E9),
   dual,
   custom(TLBATTR_QUEUEABLE,0)
   pointer_default(unique)
]
__interface IShipment: IDispatch
{
   [id(1)] HRESULT ShipOrder([in]DWORD
dwOrderNumber);
};
```

Before compiling your code, ATL 7 feeds your sources to a special precompiler that parses the attributes and generates conventional, nonattributed ATL source files, including an IDL file. The new sources are then fed to the conventional C++ compiler. In that process, the `TLBATTR_QUEUEABLE` custom attribute is converted to the IDL*QUEUEABLE* extension.

8.6 Receiving Output from a Queued Component

Banishing all output options for a queued component would be too draconian and impractical. Sometimes expecting output from a queued component is appropriate. Consider the online shoe store. After the Shipment object ships the order, you would like it to notify another object about it. The Store object would like to pass in the Notification object as a parameter to the `ShipOrder( )` method. From a design perspective, it is a good idea to decouple the notification actions from the Shipment process itself. You should decouple the action to ensure that the Shipment object knows only about a generic notification interface and that it calls a method on it once the order is shipped. It is up to the Store object to decide which notification object to pass in as a parameter. You could have multiple implementations of the Notification interface—for example, one Notification object sends an email to the customer to say that the shoes are on the way and another also sends promotional material.

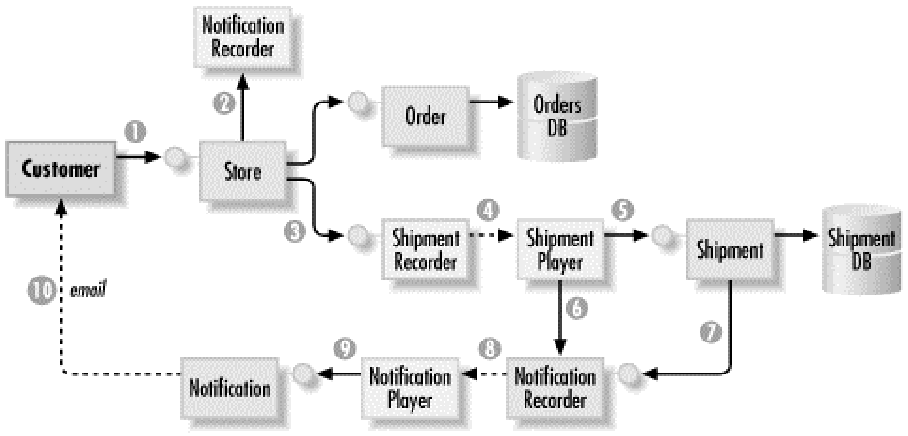
You have already seen that COM+ allows you to pass in interface pointers for objects that support the `IPersistStream` interface. COM+ also lets you pass in as a method parameter an interface pointer to another queued component. This technique is called *Response Object* because it allows the client to receive a response from the queued component and be notified about the results of the queued call. The response object does not need to support the `IPersistStream` interface, as COM+ will not try to transfer it by value to the queued component.

The client can use a response object in two ways. First, it can create the response object, create the queued component, and pass in the response object interface pointer (which actually points to the response object recorder). After the method call, the client can release the response object.

Figure 8-6 shows how a response object in the online store example is used to send notification email to the customer once the order is processed and shipped.

In the example, the customer submits the purchase order to the Store objects (Step 1). The Store object creates a Notification object (Step 2) and a Shipment object (Step 3), in both cases creating recorders only. The Store object passes in the Notification object as a parameter for the Shipment object. The Shipment recorder knows how to extract from the Notification recorder its queue name and location, and packs it in the message (Step 4). When the method call is played back to the Shipment object (Step 5), based on the information in the message, the player creates a notification recorder (Step 6) and passes it as a method parameter to the Shipment object. The Shipment object calls methods on the Notification recorder (Step 7). Once released, the notification recorder posts the queued calls to the Notification queue (Step 8), where they are eventually played back to the Notification object (Step 9). In this example, the Notification object then notifies the customer about the shipment by sending him email (Step 10).

Figure 8-6. Online store example: using a response object to send notification email



The second way a client can use a response object is to pass in string method parameters that instruct the queued component how the response object should be created. In the example, the Store object would create only the Shipment recorder and pass in as parameters where and how the Shipment object should create the Response object (machine and queue name, authentication level, and so on). The Shipment object would use these parameters as arguments for the moniker to create the Notification object.
Passing in a queued component as a parameter is more transparent to both the client and the queued component and does not contaminate the interface with parameters, which expose execution location and invocation mode. However, passing in the response object queue information provides ultimate flexibility for the client controlling the response object.
Error handling is another use for response objects. The client has no way of knowing about the success of the queued call. Imagine, for example, that while processing the order, the Shipment object was unable to complete it successfully; perhaps the vendor ran out of shoes in the requested color, or the customer supplied an expired credit card number.

The Shipment object cannot possibly handle all the error cases correctly. However, it can notify the response object that the order processing failed. The response object notifies the customer—perhaps requesting the customer to select a different color or supply a new card number. Error handling is the subject of the next section.

8.7 Queued Component Error Handling

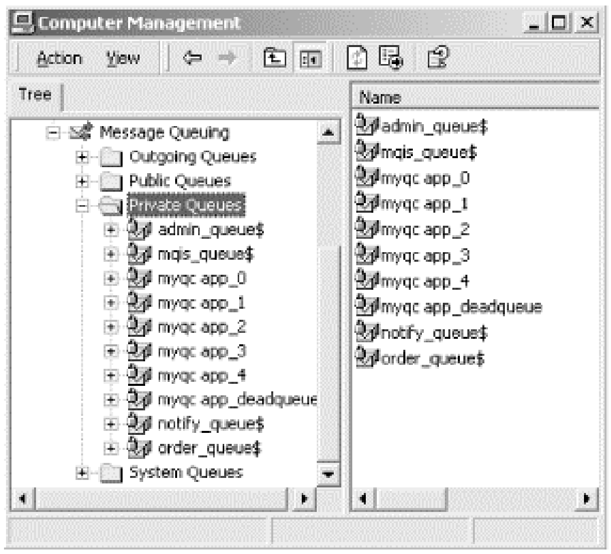
In classic synchronous COM, the client knew immediately about the success or failure of a method call by inspecting the returned `HRESULT`. A queued component client interacts with the recorder only, and the returned success code only indicates the success of recording the call. Once recorded, a queued component call can fail because of delivery problems or domain-specific errors. COM+ provides a few options for handling errors on both the client side and the server side. These options include an exception-like mechanism, auto-retries, and a few administrative tools. You can always use a response object to handle errors, as well.

8.7.1 Handling Poison Calls

Once a call is placed successfully in the application queue, plenty can still go wrong; perhaps the component was removed, its installation was corrupted, or the component failed while executing the call (for example, if the customer provided a bogus credit card number). In case of failure, if the call is simply returned back to the queue, COM+ could be trapped in an endless cycle of removing the call from the queue—trying to call the component, failing, placing it back in the queue, and so on. COM+ would never know when to stop retrying—perhaps the call could succeed the next time.
This scenario in distributed messaging systems is called the *poison message* syndrome because it can literally kill your application.
COM+ addresses the poison message syndrome by keeping track of the number of retries and managing the interval between them.
Besides creating the application public queue (where the calls are placed), COM+ creates five private *retry queues* and one *dead queue* when you enable queuing for a COM+ application (seeFigure 8-7). The dead queue is the final resting place for a message for which all delivery attempts have failed—it is a suspected poison message.
When a call is posted to a queued component, it is placed in the application public queue and a player tries to invoke it.
If the invocation fails, the message is put into Queue 0. COM+ tries to process the message again three times from queue 0, with a one-minute interval between retries. If the call still fails, the

message continues to move up the retry queues, where COM+ retries three times from each queue, with ever-increasing intervals between the retries. The higher the number of the retry queue, the longer the interval between retries (Q_1 is 2 minutes, Q_2 is 4, Q_3 is 8, and Q_4 is 16). After the last attempt from the last retry queue fails, COM+ puts the message in the dead queue, from which there are no more retries, and the message is deemed poisonous.

Figure 8-7. COM+ application private retry queues and dead letter queue



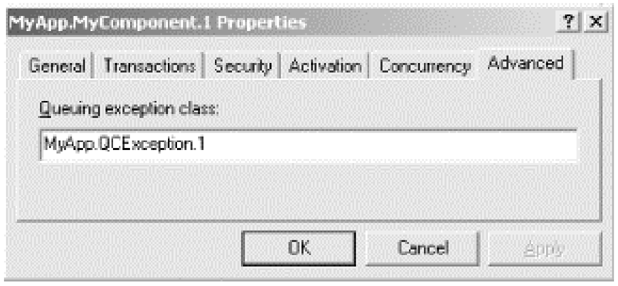
The dead queue can accumulate an infinite number of messages. It is up to your application administrator to manage the dead queue. One simple course of action available to the administrator is to simply purge the queue of all the messages it contains. Another option is to put the message back in the application or retry queues, if the administrator believes that the call will succeed this time. Your application administrator can also delete one or more of the retry queues and by doing so control the number and length of the retries; COM+ continues to move a message that continuously fails up the remaining retry queues. If all retry queues are deleted, a message that fails will be moved directly to the dead queue.

8.7.2 Queued Component Exception Classes

Sometimes it may not be possible for the call to succeed due to domain-specific constraints. For example, a customer may attempt to withdraw money from an account that has insufficient funds, or the customer account may close when the message is in the queue. Or, plain security settings may be a problem—the user who issued the queued call simply does not have the right credentials to carry out the call.

If the situation is brought to your product administrator's attention (on the client and the server side) he or she might be able to do something about it. COM+ lets you associate an exception class with your queued component. In case of repeated failure, COM+ creates the exception class object and notifies it about the failure. You associate an exception class with your queued component on the Advanced tab of your component's properties page by specifying the prog-ID of the exception calls (see Figure 8-8). If a queued call cannot reach your component, COM+ instantiates the exception class and lets it handle the failure.

Figure 8-8. Specifying an exception class for a queued component


MyApp.MyComponent.1 Properties

Queuing exception class:

MyApp.QCException.1

A queued component exception class is a COM+ component that implements all the queued interfaces supported by your component and a special interface called `IPlaybackControl`. COM+ uses the exception class object if the call could not be delivered to the queued component, or if the call persistently failed.
`IPlaybackControl` has only two methods and is defined as:

```
interface IPlaybackControl : IUnknown
{
   HRESULT  FinalClientRetry(  );
   HRESULT  FinalServerRetry(  );
};
```

The terms *client* and *server* are defined loosely in the interface. It really refers to which side of the queued call the error occurred on. Both the client and the server administrators can install the exception class, although each will be more interested in what happened on their side.

8.7.2.1 Client-side exception handling

Delivering a message to the queued component queue is never guaranteed. If all attempts to deliver the message to the queued component queue fail, COM+ places the call on the client side in a public queue called the *Xact Dead Letter queue.* The Xact Dead Letter queue is shared by all clients on the same machine.
The dead letter queue has a listener associated with it called the *Dead Letter Queue Monitor* (DLQM)—a COM+ server application installed on every Windows 2000 machine. You can start the DLQM application manually or programmatically by calling into the COM+ Catalog. When the DLQM app is running, and it detects the message

in the queue, it retrieves the target component from the message and checks for an exception class.

If the component has an exception class associated with it, the DLQM instantiates the exception class and queries it for `IPlaybackControl`. Since this is a client-side failure, the DLQM calls `IPlaybackControl::FinalClientRetry( )` on the exception class, letting it know that client-side failure of delivery is the reason it is invoked.

Next, the DLQM plays back all method calls from the message to the exception class. Recall that the exception class is required to implement the same set of queued interfaces as the component it is associated with.

If the exception class returns a failure status from any one of the method calls, the message is returned to the Xact Dead Letter queue. The DLQM deletes the message from the Xact Dead Letter Queue only if the exception class returns `S_OK` on all calls.

This error-handling schema allows the exception class to implement an alternative behavior for messages that cannot be sent to the server. For example, the exception class could generate a compensating transaction. Another course of action would pass in a queued notification object as a method parameter. The exception class would call the notification object, letting it know which calls failed. The notification object can in turn send an email to the customers asking them to resubmit the order, or it can take some other domain-specific error handling action.

Because all COM+ knows about the exception class is its ID, you can even provide deployment-specific exception classes and have a per-customer error handling policy.

8.7.2.2 Server-side exception handling

Successful delivery of the message to the server side does not mean that the call will succeed—it could still fail for domain-specific reasons, including invalid method parameters, corrupt installation, and missing components.

As explained before, the message moves up through the retry queues in case of repetitive invocation failures. When the final retry on the last retry queue fails, COM+ retrieves the target component from the message and checks for an exception class. Similar to its handling of the failure on the sending client side, if the component has an exception class associated with it, COM+ instantiates the exception class, queries for `IPlaybackControl`, and calls `IPlaybackControl::FinalServerRetry( )`. It does this to let the exception class know that the failure took place on the server side and that the message is about to be placed in the dead queue.

COM+ then plays back all method calls from the message to the exception class. The exception class can do whatever it deems fit to handle the error, from sending an email to the application administrator to alerting the user. If the exception class returns S_OK from all method calls, COM+ considers the message delivered. If the exception class returned failure on any of the queued calls, COM+ puts the message in the dead letter queue.

8.7.2.3 The MessageMover class

Regardless of where the error took place (sending or receiving side), your system or application administrator has to deal with it. Application administrators usually do not develop software for a living and know nothing about COM+, queued components, MSMQ retry queues, etc. It is up to you, the enterprise application developer, to provide your application administrator with tools to manage your product. You should deliver your main product with an application-oriented administration utility to manage retries of asynchronous calls and dead calls (on the server and client side). The application administration utility should use, in its user interface, terminology from the domain at hand (such as shipment details) rather than queue names. Internally, it will probably interact with exception classes and notification objects. Your helper utility will probably need to move messages between retry queues, the dead queue, and the application queue.
For example, suppose a queued call destined for a customer accounts management component failed because the specified account number was invalid. The administration utility may prompt the administrator to enter the correct account number for the customer and then put the message back in the application queue, this time with the correct account number. To enable you to move messages between queues easily, COM+ provides you with the IMessageMover interface and a standard implementation of it. The standard implementation is available for the C++ developer by calling CoCreateInstance( ) using CLSID_MessageMover and for the Visual Basic developer by calling CreateObject( ) using the prog-ID QC.MessageMover.
The interface IMessageMover is defined as:

```
interface IMessageMover : IDispatch
{
   [id(1),propget] HRESULT SourcePath([out,retval]BSTR*
pbstrPath);
   [id(1),propput] HRESULT SourcePath([in] BSTR
bstrPath);
   [id(2),propget] HRESULT DestPath([out,retval] BSTR*
pbstrPath);
   [id(2),propput] HRESULT DestPath([in]BSTR bstrPath);
   [id(3),propget] HRESULT
CommitBatchSize([out,retval]long* plSize);
```

```
   [id(3),propput] HRESULT CommitBatchSize([in]long
lSize);
   [id(4)] HRESULT MoveMessages([out, retval]long*
plMessagesMoved);
};
```

As you can see, `IMessageMover` is a simple interface. You can set the source and destination queues and call `MoveMessages( )`, as shown in Example 8-2, in Visual Basic 6.0.

Example 8-2. Using the IMessageMover interface to move messages from the last retry queue to the application queue

```
Dim MessageMover  As Object
Dim MessagesMoved As Long

Set MessageMover = CreateObject("QC.MessageMover")

'move all the messages from the last retry queue to the
application queue
MessageMover.SourcePath = ".\PRIVATE$\MyApp_4"
MessageMover.DestPath   = ".\PUBLIC$\MyApp"

MessagesMoved = MessageMover.MoveMessages
```

`IMessageMover` does not provide you with a way to move fewer than the total number of messages on the queue, but it does save you the agony of interacting directly with the MSMQ APIs.
See the MSDN Library for more information about using the `IMessageMover` interface.

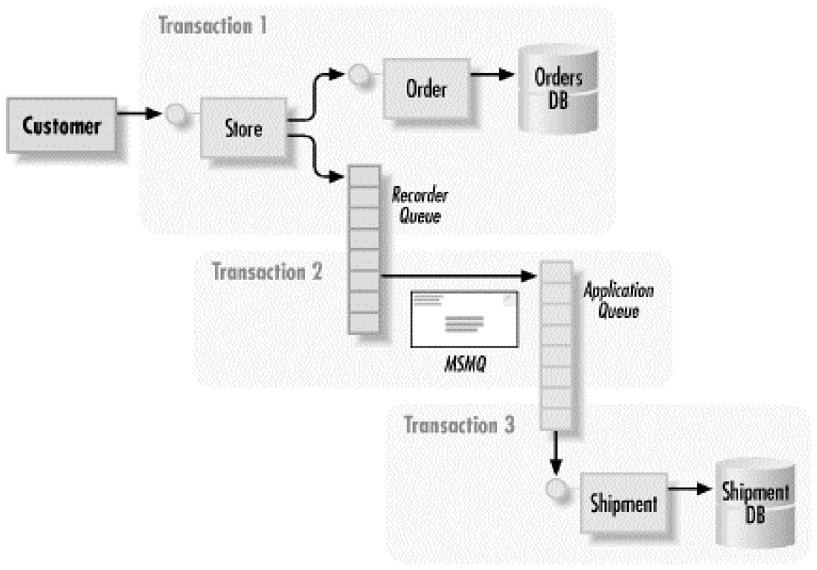
8.8 Queued Components and Transactions

As mentioned before, MSMQ is a resource manager. By default, when COM+ creates the application and retry queues, they are all transactional queues; they auto-enlist in the transaction that adds or removes a message to or from the queue.
The recorder and the listener are COM+ components installed in the COM+ Utilities application—a library application that is part of every Windows 2000 installation. These components are configured to require a transaction and take part in an existing transaction, or spawn a new one if none exists. (COM+ will not let you change the Utilities application components settings). Every time a client uses queued components, three transactions are involved—recording the calls, delivering the message to the application queue, and replaying the calls.
You can see this concept work with the online store (see Figure 8-9); all the calls made by the Store object on the Shipment recorder are packaged into one message and placed in an intermediate recorder queue. These calls were made in the scope of the

transaction that accepted the order parameters from the customer (Transaction 1). Since MSMQ is a resource manager, the recorder queue rolls back and rejects the message if the order transaction is aborted.

MSMQ then has to transfer the message to the queued component application queue, potentially across the network. MSMQ creates a new transaction for the transfer, and both the source and the destination queues participate in it. If the transfer was unsuccessful, the transaction aborts, the queues roll back, and the message remains in the recorder queue. This action avoids a partial success situation, in which the message is removed from the source queue, but never added to the destination queue. This transaction is called Transaction 2 in Figure 8-9.

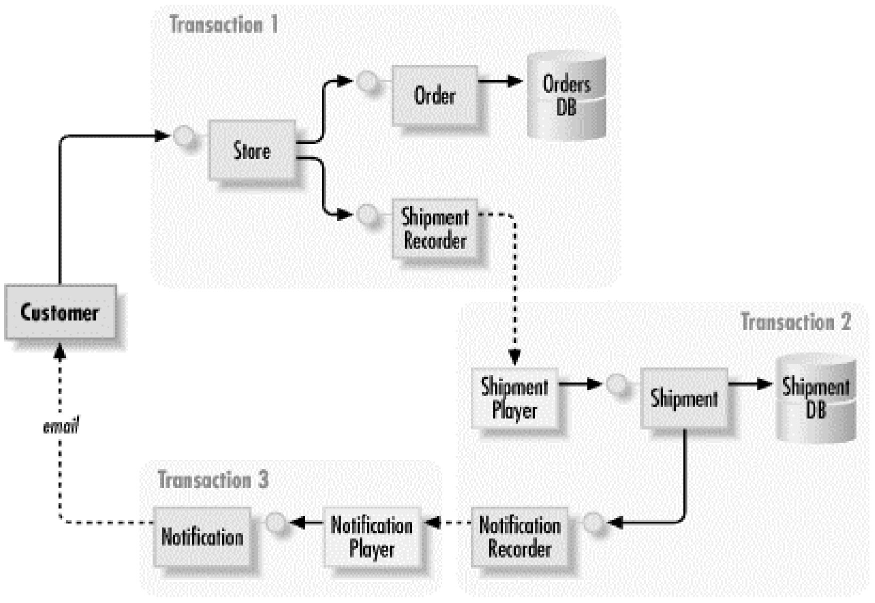
Figure 8-9. The three transactions involved when a client uses queued components



Once the message is safely in the application queue, the listener starts a new transaction for removing it and playing it back to the component (called Transaction 3 in Figure 8-9). If the invocation fails, the application queue rollback moves it to the first retry queue, instead of adding it back to the application queue, to detect a potential poison message.

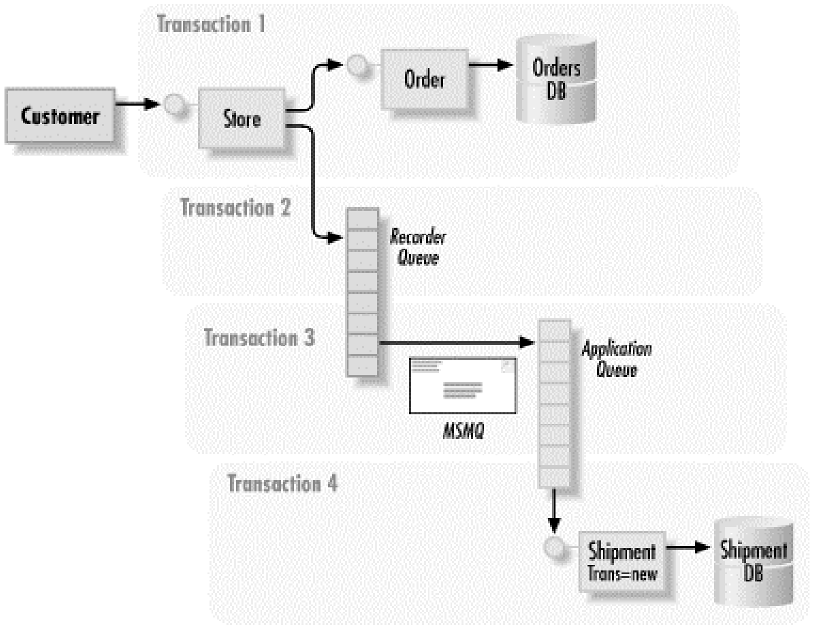
Usually you take the MSMQ transfer transaction for granted and omit it from your design documents. If you use a response object, the response object playback would be in a transaction of its own because it is just another queued component (see Figure 8-10).

Figure 8-10. The transaction involved when using a response object



A word of caution when configuring the transactional setting of a queued component: avoid configuring your queued component to require a new transaction of its own. If you configure your component's transaction setting to have Requires New, the recorder is in a separate transaction from that of the client, and MSMQ accepts the recorded calls and posts them to the application queue even if the original client transaction fails (see Figure 8-11).

Figure 8-11. Avoid configuring a queued component to require a new transaction



A similar inconsistency may exist if you configure the application queue as a nontransactional queue, as MSMQ removes the message from the queue even if the Shipment transaction is aborted.

You should always set the transaction setting of your queued component to Required—that is what will be necessary in most business situations.

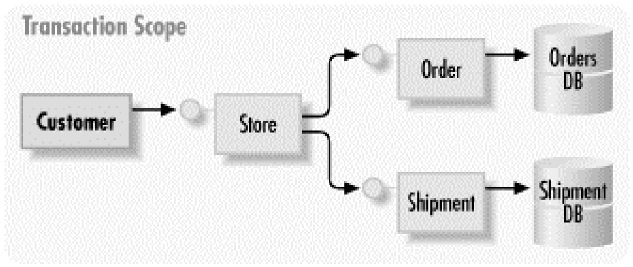
8.9 Synchronous Versus Asynchronous Components

By now you have probably come to appreciate the elegance of using queued components and the great ease with which you can turn a synchronous component and its client code to asynchronous. However, although it is technically possible to use the same components both synchronously (using `CoCreateInstance( )` to create it) or asynchronously (using the `queued` moniker), the likelihood that a component will be used in both cases is low for the following reasons: using a queued component introduces changes in the semantics of the transactions the component will take part in, and using a queued component implies a change in the client program workflow. You simply cannot use the same synchronous execution sequence logic. The rest of this section elaborates on these two reasons. These arguments were first presented in Roger Session's book *COM+ and the Battle for the Middle Tier* (John Wiley & Sons, 2000).

8.9.1 Changes in Transaction Semantics

Suppose your online store does not use queued components. When the customer places an order, the Store object uses the Order and the Shipment objects synchronously. All the order and shipment database updates that these objects perform are under the umbrella of one transaction. Both databases are consulted regarding committing the transaction (see Figure 8-12).

Figure 8-12. Synchronous invocation scopes all operations in one transaction



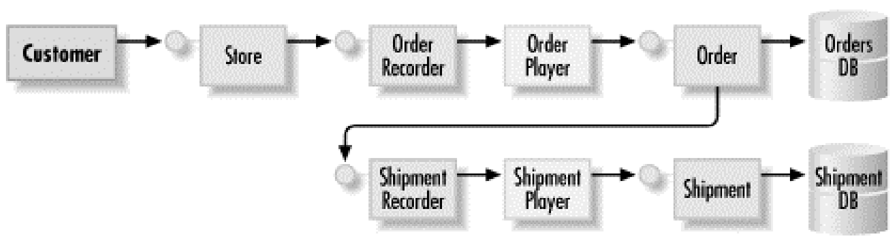
However, if the Store object uses a queued Shipment component, as shown in Figure 8-9, the shipment database and component are not part of the originating transaction and are not consulted regarding its success. The Shipment transaction is now entirely different from the Order transaction. By the time the shipment transaction takes place, the order transaction has already been

committed. Even if the shipment transaction aborts, the order transaction remains committed and its changes will not roll back. Of course, the shipment transaction may retry and eventually succeed and commit, and that may be fine. On the other hand, it may always fail, and that is probably not so fine.
The conclusion is that configuring a component to be asynchronous has serious implications regarding the semantics of the transactions it participates with.

8.9.2 Changes in Workflow

The other major difference between working with a queued component as opposed to its nonqueued version has to do with the client workflow. Currently, your Store object calls the Order object synchronously, and only if the Order object succeeds in processing the order will the Store object call the Shipment queued component. Suppose the Store object would like to use a queued version of the Order component (besides a queued Shipment component). The Store object records the calls to the Order component, records the calls to the Shipment components, and then releases the recorders.
The problem is that the Order and Shipment objects might be invoked in random order, depending on the network topology, MSMQ setup, number of retries, and so on. The result can be disastrous if things go wrong—for example, if the Shipment object discovers that no shoes in the store match the customer request, but the Order object has already billed the client for it.
Again, you will find that you cannot just configure components as queued and use them asynchronously since doing so results in potentially flawed workflow.
Using a queued component instead of a synchronous version of the same component requires you to change your code and your workflow. If the Store component developer wants to use both queued Order and queued Shipment components, the Store object should only call the queued Order component. To avoid the potential inconsistencies mentioned earlier, the call to the Shipment queued component should be done by the Order object only if the order processing was successful (see Figure 8-13).

Figure 8-13. Queued Order and Shipment components require changing the application workflow



In general, if you have more than one queued component in your workflow, you should have each component invoke the next one in the logical execution sequence. Needless to say, such a programming model introduces tight coupling between components (they'll have to know about each other) and changes to their interfaces, since you have to pass in additional parameters required for the desired invocation of queued components down the road.
In addition to changes in the workflow and interfaces, you still face the problem of having the order and shipment operations in separate transactions. The only way to have them share the same transaction is to make them synchronous.
The conclusion from this simple example is that using the asynchronous version of a component instead of its synchronous version introduces major changes to the component interfaces, the client workflow, and the supporting transaction semantics. A queued component should be designed for queuing from the ground up. The handy "Queued" checkbox is merely configuration sugar on top.

8.10 Queued Components Security

As you saw in Chapter 7, security is an essential part of any distributed application, and COM+ provides you with a rich, user-friendly security infrastructure. When a client makes a queued call, the queued component may still require the same level of security services and protection as if it were invoked synchronously, and rely on COM+ to provide authentication and authorization.
However, the underlying method call invocation is different, and the synchronous security mechanism simply will not do—by the time the actual object is invoked, the client may be long gone (with its security identity and credentials). The synchronous authentication that uses the challenge-response mechanism cannot be used.
The idea behind queued component security is simple—have the recorder capture the security identity (and other security-related information) of the client as it records the method calls. The security information is bundled in the message along with the method calls and sent to the queued component application queue. Before the player makes the call on the component itself, COM+ uses the captured security information to validate that the client is allowed to access the component.
The underlying implementation of this idea relies heavily on MSMQ security services to capture the client security details in the message and transfer it securely to the application queue. To ensure authenticity of the message, the messages can carry a digital signature from the client. MSMQ can even encrypt the message for transfer. If, on the receiving side, MSMQ encounters a message with insufficient security credentials or a message that

was tampered with, then MSMQ puts it in the application's dead queue.

8.10.1 Queued Calls Authentication

The default call authentication level actually depends on the queued component application settings. If the application uses role-based security, then during the call to `CoGetObject( )`, COM+ captures the information required to authenticate the call during playback in the message. The queued component client can explicitly specify the desired authentication level for the queued call and the required privacy level by providing the queued moniker with additional parameters.

If the client requires authentication, MSMQ digitally signs the message with the user's security certificate. If this is the case, your application administrator has to issue an MSMQ security certificate for each potential user by using the MSMQ administration applet in the Control Panel.

In Example 8-3, the Store object explicitly turns on authentication and instructs MSMQ to encrypt the message body.

Example 8-3. Explicitly setting authentication and encryption levels for a queued call

```
IShipment* pShipment = NULL;
HRESULT hres = S_OK;
DWORD dwOrderNumber = 123;
hres = ::CoGetObject(L"queue:AuthLevel=
MQMSG_AUTH_LEVEL_ALWAYS,
                        PrivLevel= MQMSG_PRIV_LEVEL_BODY
                        /new: MyApp.Shipment", NULL,
                        IID_IShipment,(void**)&pShipment);

hres = pShipment->ShipOrder(dwOrderNumber);

pShipment->Release(  );
```

Exceptionally paranoid clients can also specify the encryption algorithm to use and a cryptographic hash function (see the MSDN Library for details on these advanced parameters for the queue moniker).

Insisting on high security carries with it the usual performance/security tradeoff. Decide on your security setting wisely. For example, you may want to authenticate only the actual order shipment call, but not less sensitive method calls.

8.10.2 Queued Components and Role-Based Security

Despite the fact that under-the-hood COM+ uses a drastically different mechanism for queued components security, the queued

component can, once instantiated, take advantage of role-based security with the same ease as if it were invoked synchronously. You can configure your component administratively to use role-based security on the component, interface, and method levels, and even use programmatic role-based security calls such as `IObjectContext::IsCallerInRole( )`.
The only requirement for using role-based security is that the call be authenticated. If the client explicitly turns authentication off while role-based security is in use, the call will fail during playback, since COM+ has no way of authenticating what role the client belongs to.

8.10.3 Queued Components Security Limitations

A queued component developer has access to similar security features and services as a nonqueued component, and from a security standpoint, your code will be the same as if you were developing a normal synchronous component. However, some differences do exist, especially if you plan to use the more advanced or esoteric security services. You should be aware of the following limitations:

- The queued component developer is discouraged from performing low-level security manipulation, such as interacting directly with the Kerberos authentication service, because the Kerberos cookies are not part of the MSMQ message. Generally, if you want to do low-level security calls, you are restricted to whatever MSMQ supports.
- Queued components do not support impersonating the client. This is done (by design) to close a potential security hole in which an untrustworthy source has generated a message whose format resembles that of a message to a queued component and placed it in the application queue. COM+ requires the original client’s security identity to compare with the message sender identity to verify that the message came from the client. By doing so, however, COM+ inhibits impersonation.
- If you install MSMQ using the Workgroup configuration, MSMQ cannot authenticate queued calls. As a result, you should turn off security access checks at the application and component levels.

8.11 Queued Components Pitfalls

Queued components is a great service, no doubt, but it does have a few quirks and pitfalls that I would like to point out.

8.11.1 MSMQ Setup

MSMQ can be installed in two configurations. The first relies on having a Windows 2000 domain server present on the network. The workstation onto which you wish to install MSMQ must be part of that domain. The second installation option is for a Windows Workgroup.
To call queued components across the network securely, queued components require the presence of a Message Queuing Primary Enterprise Controller (PEC) on the network. If you install MSMQ for Workgroup, you have to turn the security knob all the way down (set the authentication level for the queued components application to None and avoid using access control checks). Any cross-machine calls must be unauthenticated. This limitation is serious. For any Enterprise-level worthy application, you need the MSMQ domain server installation.

8.11.2 Queued Component Client

A client of a queued component can run only on a Windows 2000 machine. There is no apparent reason for this condition, as every Microsoft platform supports MSMQ. What makes it even more awkward is the fact that most portable devices that could benefit from disconnected sessions will not run Windows 2000.[1]

[1] I can only say that I find this situation very strange, and I hope that Microsoft will amend this predicament soon.

8.11.3 Visual Basic Persistable Objects

As mentioned before, a queued component client can pass in as a method parameter an interface pointer to a COM object, provided that the object supports the `IPersistStream` (so that COM+ can serialize the object state into a stream).
However, if the object is written in Visual Basic 6, the object must be initialized before making the call on the recorder interface by querying it for `IPersistStream` and calling one of the `IPersistStream` methods `Init( )`, `InitNew( )`, or `Load( )`.
If your client is written in Visual Basic as well, the Visual Basic runtime handles the object initialization automatically for you. If the client application is written in C++, the application must initialize the component explicitly. Requiring the client to know the language used to implement the queued component couples the client to the

component, but knowing of a limitation is better than trying to figure out what went wrong.

8.11.4 IDispatch Considerations

When a queued component client makes a call, it actually interacts with a recorder. The recorder has to match as much as possible the behavior of the real component, including its implementation of `IUnknown::QueryInterface( )`. The recorder bases everything it does on the component-type library. It is common for a component to support multiple interfaces derived from `IDispatch` . If that is the case, what interface should the recorder return to the client when it is queried for `IDispatch( )`?
The recorder uses the following algorithm to provide the right `IDispatch( )`:

- If the component default interface inherits from `IDispatch`, the default interface is returned.
- If no interface is marked as default, but only one interface inherits from `IDispatch`, that interface is returned.
- If no interface is marked as default and multiple interfaces inherit from `IDispatch`, the recorder returns `E_NOINTERFACE`.

The obvious recommendation is to always mark one of your component `IDispatch`-derived interfaces as the default interface.

8.11.5 Queued Component Application Startup

When an application hosts queued components, COM+ must activate a listener for queued calls sent to its queue whenever the application is launched. If you package queued and nonqueued components in a single application, the application might service clients of nonqueued components when a queued call arrives. This situation may be a cause for concern if the queued component makes a lot of CPU-intensive calculations or requires other expensive resources. These characteristics may be the reasons you made that component queued—so that your component will not be in the way of other components and will do the expensive processing at times when the system load is low.
When deciding on component allocation to applications, make sure that you really want queued components to start when a nonqueued component executes. If you would like to control the queued components' execution time, package the queued components into a separate COM+ application and explicitly start it up when you deem it fit.

8.12 Summary

Component developers benefit from COM+ queued components on different levels. First, they take away the need to handcraft asynchronous method invocation solutions, allowing developers (as with the other COM+ component services) to focus on the business problem at hand. Second, and perhaps just as important, queued components' ability to take seamless advantage of other COM+ services, such as transactions and role-based security, is something that would be almost impossible to provide in a custom solution. You can even combine queued components with COM+ loosely coupled events, the subject of the next chapter.

Chapter 9. COM+ Event Service

In a component-oriented program, an object provides services to clients by letting clients invoke methods on the object's interfaces. But what if a client (or more than one client) wants to be notified about an event that occurs on the object side? Traditionally, the client implements a callback interface called a *sink interface*. To notify the client of an occurring event, the object calls a method on the sink interface. Each method on a sink interface corresponds to a type of event fired by the object.

This model raises a few questions: How does the object access the sink interfaces? How do clients find out which sink interfaces the object fires events on? How do the clients unsubscribe from event notification?

As you will see shortly, the COM+ events service is an exciting new service that evolved to address the classic problems of event notification and reception. COM+ events are also known as *Loosely Coupled Events* (LCE), because they provide an effective way of decoupling components. They put the logic for publishing and subscribing to events outside the scope of the component. Besides significantly improving on the classic COM model for handling events, LCE takes full advantage of such COM+ services as transactions, queuing, and security. Managing event publishing and subscription can be done both declaratively via the Component Services Explorer and programmatically using the COM+ Catalog.

To fully appreciate the elegance of the COM+ events service, you should first understand the drawbacks of the way classic COM handles events.

9.1 Classic COM Events

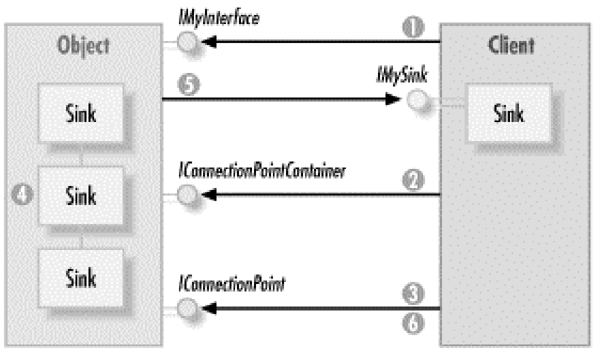
In classic COM, when a client wants to receive events from an object, the client has to pass the object an interface pointer to a client implementation of the sink interface. This operation is called *advising* the object of the sink. Advising takes place by either using a standard mechanism (*connection points*) or a custom one very similar in nature. These mechanisms have changed little since the early days of OLE 2.0 and COM.

If you use connection points, the object has to implement an interface called `IConnectionPointContainer` (see Figure 9-1). The client uses the connection point container interface to find out whether the object supports firing events on a specified sink interface IID. Think of it like a kind of reverse `QueryInterface( )` call: the client queries the object for its ability to use an interface implemented by the client.

Establishing a connection point usually follows a pattern similar to this one:

1. The client queries an existing object interface for `IConnectionPointContainer`.
2. The client uses `IConnectionPointContainer` to find out whether the object supports firing events on a specified sink interface. If it does, the object returns to the client an object-side implementation of another interface called `IConnectionPoint`.
3. The client uses `IConnectionPoint` to advise the object of the client-side sink interface.
4. The object has to maintain a list of sinks that have subscribed. It adds the new sink to the list and returns to the client a cookie identifying the connection. Note that the object manages the subscription list.
5. The object uses the sink interface to notify the client(s) about events.
6. When the client wants to stop receiving events and break the connection, it calls `IConnectionPoint::Unadvise( )`, passing in the cookie that identifies the connection.

Figure 9-1. Classic COM managed events using connection points



Establishing the connection requires expensive round trips, potentially across the network. The client must repeat this cumbersome sequence for every sink interface on which it wants to receive events and for every object from which it wants to receive events. Using connection points, there is no way for the client to subscribe to a subset of events the object can fire. The client has no way of filtering events that are fired (Notify me about the event only if...); as a result, a COM designer often opts for the use of a custom mechanism (instead of the generic connection points) that allows subscription to a subset of events. Needless to say, this solution introduces coupling between the object and its clients regarding the specific interaction.
Connection point clients must also have a way to access a server instance (the object) to advise it of the sink. Usually the clients

know the server CLSID, get the object from another client, or go through some initialization protocol. That, in turn, also introduces coupling between clients and objects and coupling between individual clients.
On the object side, the object has to manage a list of sinks. This code has almost nothing to do with the domain problem the object is designed to solve. Properly managing the list of sinks requires marshaling sink pointers to a worker thread manually to actually perform event firing. That extra code introduces bugs, testing time, and development overhead. To make matters worse, the same code for managing connections is repeated in many servers.
With this model, the object and the clients have coupled lifetimes—the server usually AddRefs the sinks and the clients have to be running to receive events. There is no way for a client to say to COM "If any object fires this particular event, then please create an instance of me and let me handle it."
There is no easy way to do disconnected work—that is, the object fires the event from an offline machine and the event is subsequently delivered to clients once the machine is brought online. The reverse is also not possible—having a client running on an offline machine and receiving events fired while the connection was down.
Setting up connections has to be done programmatically. There is no administrative way to set up connections.
The events, like any other COM call, are synchronous. The object is blocked while the client handles an event. Other clients are not notified until the current client returns control back to the object. Well-behaved clients avoid lengthy processing of the events (by perhaps delegating to a client-side worker thread), but there is no way of forcing clients to behave nicely or to fire the events on multiple threads without writing a lot of complex code.
There is no safe way to mix transactions and events. Suppose an event fires, but then the transaction the object took part in is subsequently aborted. How can the object notify the clients to roll back?

9.2 COM+ Event Model

The COM+ event model is based on a simple idea—put the connection setup and the event firing plumbing outside the scope of the components. Under COM+, an object that fires events is called a *publisher*. A client who wants to receive events is called a *subscriber*. Subscribers who want to receive events register with COM+ and manage the subscribe/unsubscribe process via COM+, not the object. Similarly, publishers hand over the events to COM+, not directly to the subscribed clients.

COM+ delivers an event to the clients that have subscribed. By having this layer of indirection, COM+ decouples your system. Your clients no longer have any knowledge about the identity of the publishers. The subscription mechanism is uniform across all publishers, and the publishers do not manage lists of connections. The rest of this chapter explains the details of the COM+ events service, its capabilities and limitations, and its interaction with other COM+ services.

9.3 The Event Class

A publisher object fires an event at COM+ (to be delivered to the subscribers) using an *event class*. The event class is a COM+ provided implementation of the sink interfaces the publisher can fire the events at. The implementation is synthesized by COM+, based on a type library you provide. This library contains the interface definitions and stipulates which CoClass implements them. COM+ uses the same CoClass definition for its implementation of the event classes. To publish an event, the publisher first CoCreates the event class (the publisher has to know the event class CLSID) and then fires the events at its interfaces.
For example, suppose an object wants to fire events at the sink interface `IMySink`, using an event class called `MyEventClass`. `IMySink` is defined as:

```
interface IMySink : IUnknown
{
   HRESULT OnEvent1(  );
   HRESULT OnEvent2(  );
};
```

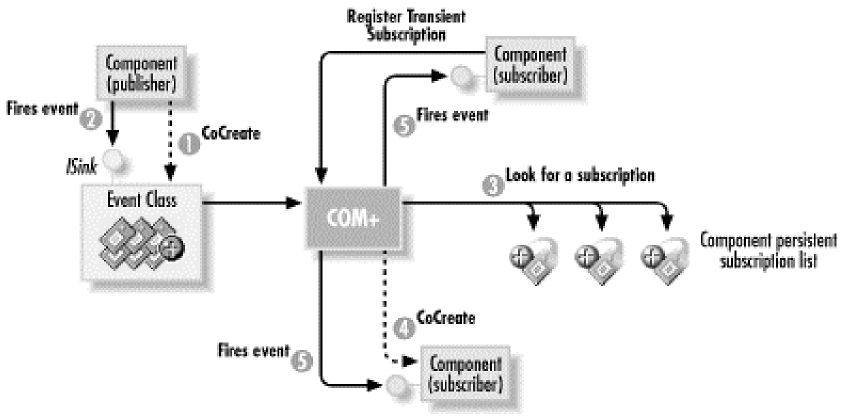
The publisher code looks like:

```
HRESULT hres = S_OK;

IMySink* pMySink = NULL;

hres =:
=:CoCreateInstance(CLSID_MyEventClass,NULL,CLSCTX_ALL,IID
_IMySink,
                               (void**)&pMySink);
ASSERT(SUCCEEDED(hres));

hres = pMyEvent->OnEvent1(  );
ASSERT(hres == S_OK);

pMyEvent->Release(  );
```

Compare the simplicity on the publisher side to classic COM connection points—the publisher does not have to manage lists of

subscribers. All the publisher has to do is create an event class and fire the event on it.
Figure 9-2 illustrates the interaction between the publisher, the event class, COM+, and the subscribers. The client creates the event class (Step 1) and fires the event at it (Step 2). When the publisher is finished with the event class, it can either release it or cache the event class interface pointer for the sake of performance, to be used the next time the publisher wants to publish events.

Figure 9-2. The COM+ event system at work

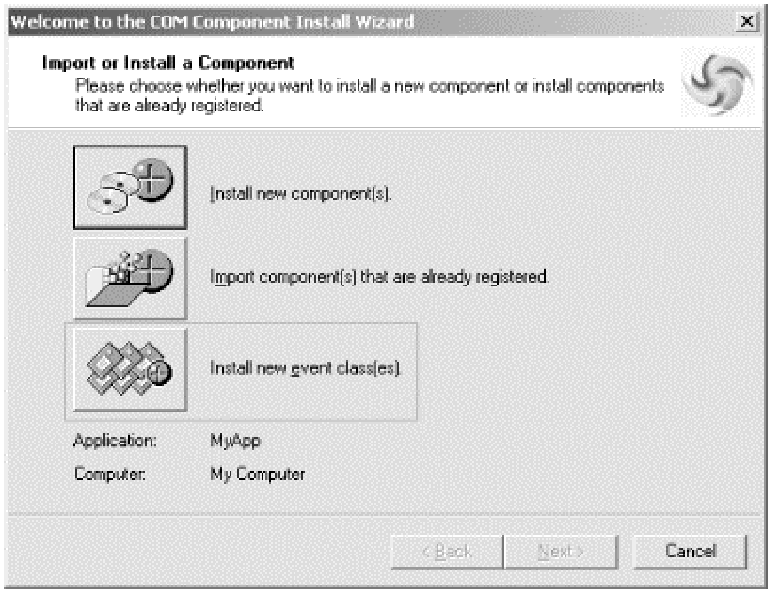


The COM+ implementation of the event class interfaces goes through the list of subscribers on that event class (Step 3) and publishes the events to them. COM+ maintains a list of subscriptions for every event class. The subscriptions can be interface pointers to existing objects (called *transient* subscriptions) or CLSID for a class (called *persistent* subscriptions).
In the case of a persistent subscription, COM+ creates an object of the type specified by the CLSID (Step 4), calls the appropriate sink method on the object (Step 5), and releases the object. In the case of a transient subscription, COM+ simply calls the appropriate sink method on the object (Step 5).
It is interesting to note that firing the event is by default serial and synchronous—that is, the subscribers are called by default one after the other (serial), and control returns to the publisher object only after all the subscribers are notified (synchronous).

9.3.1 Adding an Event Class

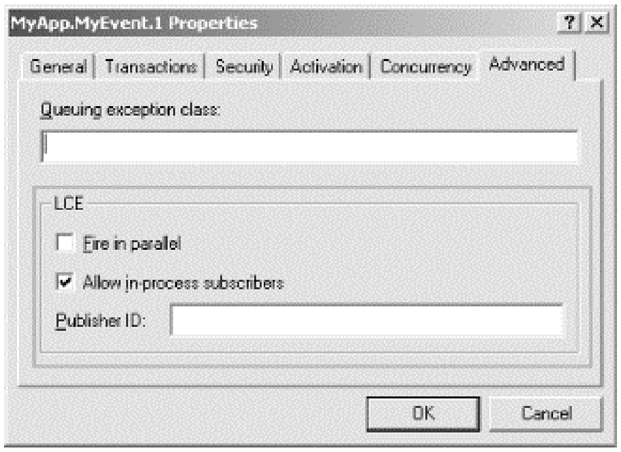
You can add an event class to the Component Services Explorer by using the Component Install Wizard. Bring up the wizard for installing a new component to your application and select Install new event class(es) (see Figure 9-3).

Figure 9-3. The Component Install Wizard is used to add a new event class


Welcome to the COM Component Install Wizard
Import or Install a Component
Please choose whether you want to install a new component or install components that are already registered.
Install new component(s).
Import component(s) that are already registered.
Install new event class(es).
Application: MyApp
Computer: My Computer
< Back
Next >
Cancel

The rest of the steps in the wizard are the same as when adding a new COM+ component. When you point the wizard at a DLL containing a type library with sink interface and event CoClass definitions (more about those in a minute), under-the-hood COM+ synthesizes its own implementation of the interfaces and installs the synthesized components instead of yours.
After installing the event class in the Component Services Explorer, the only way to detect that it is not a user-implemented COM+ component is to inspect its component properties page on the Advanced tab. The Advanced tab of an event class contains the Loosely Coupled Event (LCE) group (see Figure 9-4).

Figure 9-4. The LCE group configures event class-specific settings



Queuing exception class:

LCE

☐ Fire in parallel

☑ Allow in-process subscribers

Publisher ID:

You can add an event class component to any COM+ application, be it a library or a server application.

9.3.2 Supplying the Event Class Definition

For COM+ to implement an event class for you, you have to provide COM+ with the sink interfaces definitions, the event class CLSID, and the interface each event class supports. You provide this information in the form of a type library. The type library has to be embedded as a resource in a DLL. The Component Install Wizard knows how to read the type library from the DLL and detect the CoClass definitions inside.
For every CoClass in the type library, COM+ tries to generate an event class and add it to your application as a component. COM+ synthesizes implementation only to interfaces that are part of the event class CoClass definition in the type library.
For example, to define the event class `MyEventClass` that supports the sink interface `IMySink` (shown earlier), your IDL file should look like this:

```
[
   uuid(0A9B9E44-E456-4153-9FC8-5D72234B7C82),
   version(1.0),
   helpstring("Event Class 1.0 Type Library")
]
library EVENTCLASSLib
{
   importlib("stdole32.tlb");
   importlib("stdole2.tlb");
   importlib("SubApp.tlb");//The subscribers' TLB

   [
      uuid(5CAF8E95-3FEF-40F1-94C3-3F408240D53B),
      helpstring("MyEventClass Class")
   ]
   coclass MyEventClass
   {
      interface IMySink;
   };
};
```

To avoid repeating the definition of the sink interfaces in both the subscriber application and the event class type library, the event class IDL file should import the sink interface (`IMySink`) definitions from the type library of the subscribers. This is what the line `importlib("SubApp.tlb");`was used for in the previous example.
The easiest way to generate a type library is to have the Visual Studio ATL create one for you. The default behavior in ATL is to embed the type library in the DLL, since the ATL Application Wizard adds a reference to the type library in the project RC file.
I strongly recommend that you put only event classes in the event class DLL. Do not put event classes in the same type library with regular CoClasses; such a mix confuses the Install Wizard—the Wizard will install all components as event classes. This installation has potentially catastrophic results, since it may corrupt an existing

installation of the regular components. However, as you have already seen in Chapter 1, you can map more than one DLL to the same COM+ application—you can put your event class and other components in the same application.
When you supply the event class, COM+ tries to register it. You are responsible for providing proper registration code in the DLL for all components contained in the DLL. Again, the easiest way is to use ATL to generate a skeleton implementation of the event class for you. Simply have the ATL Object Wizard insert new components into the event classes DLL. Since the implementation of these event classes is never called, it is a bug if anybody ever uses them. This would usually happen as a result of not installing the event class in the COM+ Catalog and only building and registering it as a normal COM object. I therefore suggest that you provide default behavior to the ATL code-assert on every method call. See Example 9-1.

Example 9-1. Skeleton implementation of the event class

```
class CMyEventClass :
   public CComObjectRootEx<CComMultiThreadModel>,
   public CComCoClass<CMyEventClass,&CLSID_MyEventClass>,
   public IMySink
{
public:
   CMyEventClass(  ){};
   DECLARE_REGISTRY_RESOURCEID(IDR_MYEVENTCLASS)
   DECLARE_PROTECT_FINAL_CONSTRUCT(  )

   BEGIN_COM_MAP(CMyEventClass)
     COM_INTERFACE_ENTRY(IMySink)
   END_COM_MAP(  )

// IMySink
public:
   STDMETHOD(OnEvent1)(  ){ATLASSERT(0);return
E_NOTIMPL;};
   STDMETHOD(OnEvent2)(  ){ATLASSERT(0);return
E_NOTIMPL;};
};
```

9.3.3 Event Class Interface Design Guidelines

The sink interface can be a custom interface or an automation-compliant interface. However, the methods of a sink interface can contain only input parameters. `[out]` or `[in,out]` parameters are not allowed. Since COM+ seeks to decouple the publisher from the subscribers, there is no way for a subscriber to return information back to the publisher—the call on the subscriber interface returns to COM+, not to the publisher.

From the publisher's perspective, it only fires an event on one object—the event class.
COM+ uses type library marshaling to marshal the call on the sink interface from the event class to the subscribers. Interfaces that use type library marshaling must comply with the following requirements:

- All the methods must return `HRESULT`.
- The methods do not use certain IDL attributes such as `[size_is]` and `[length_is]`. See the MSDN documentation for the exact specification of typelib-compliant IDL attributes.

9.4 Subscription Types

As I mentioned earlier in the chapter, there are two types of subscribers. The first type is an existing instance of a class that supports a sink interface. That instance can be added at runtime to the list of subscribers of a particular event class. This type of subscription is called *transient subscription* because it exists as long as the subscriber is running and will not persist or survive a system reboot or a crash.
Note that when a particular instance of a class subscribes to an event class, only that instance will receive events published using that class. Other instances will receive the events only if they transiently subscribe themselves.
Adding a transient subscription can only be done programmatically using theCOM+ Catalog interfaces and objects. There is no administrative Component Services Explorer support. On the other hand, since all you give the COM+ Catalog is a pointer to a sink, even a nonconfigured COM component can register as a transient subscription, as long as it supports the sink interface.

> A transient subscription does not even need to be on a Windows 2000 machine, as long as it is registered with a COM+ Catalog on the Windows 2000 machine where the event class resides.

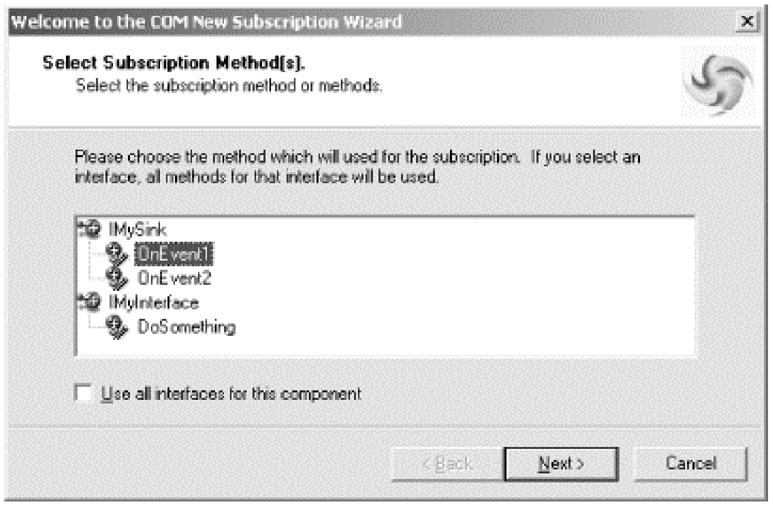
The second type of subscription is used when you want COM+ to instantiate an object of a particular class when an event is published, let it handle the event, and release it. This type of subscription is called *persistent subscription.* Every event class has a list of persistent subscribers associated with it, stored in the COM+ Catalog. Persistent subscriptions, as the name implies, persist in the COM+ Catalog and survive a system restart.
Objects created by a persistent subscription are always released after each event delivery, even if more events are on the way. As a

result, your subscribing component should be written to handle each event independently of other events that may or may not be published or delivered.

9.4.1 Adding a Persistent Subscription

Every component in the Component Services Explorer has a Subscription folder, containing the persistent subscriptions the product administrator or developer has set up. Every subscription represents an event class (or a list of event classes) that the component should be instantiated to receive events from whenever any publisher uses those event classes.
To add a persistent subscription, expand the subscription folder, right-click on it and select New from the pop-up context menu. This action invokes the New Subscription Wizard (see Figure 9-5).

Figure 9-5. The New Subscription Wizard


Welcome to the COM New Subscription Wizard

Select Subscription Method[s].
Select the subscription method or methods.

Please choose the method which will used for the subscription. If you select an interface, all methods for that interface will be used.

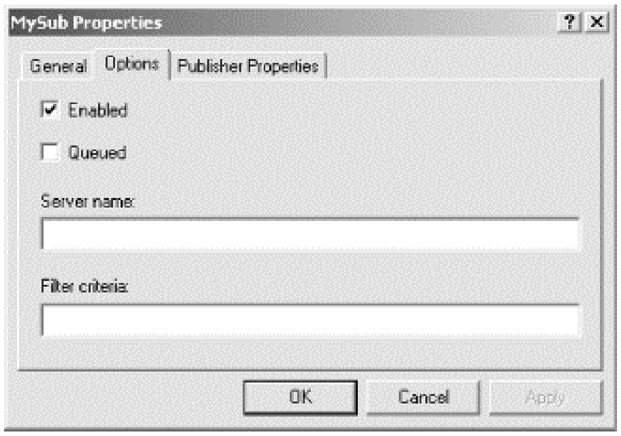
```
IMySink
  OnEvent1
  OnEvent2
IMyInterface
  DoSomething
```

☐ Use all interfaces for this component

The New Subscription Wizard lets you subscribe to events published to all the sink interfaces your class supports, to a particular interface, or even just to a particular method. The wizard displays all the interfaces your component supports, including nonsink interfaces—COM+ doesn't know whether they are sinks or not; only you know.
You can set up a subscription at the method or interface level. At the method level, COM+ delivers the event to your component only when publishers use that method. If you want to subscribe to another method, you have to add a new subscription.
A subscription at the interface level means that any event targeting any method on that interface should be delivered to your component. By providing you with these two options, you have the ability to subscribe to only a subset of the events publishers can publish or to all of them.

After you select the interfaces and methods, the wizard displays a list of all installed event classes that support the interfaces you selected in the previous steps. You can select a particular event class or all of them. The last step in the wizard lets you name the subscription and enable it. You can always enable or disable a subscription by highlighting it in the Subscriptions folder, displaying its properties page, selecting the Options tab, and enabling or disabling the subscription (see Figure 9-6).

Figure 9-6. A persistent subscription's Options tab


MySub Properties

General | Options | Publisher Properties

[✓] Enabled

[ ] Queued

Server name:

Filter criteria:

9.4.2 Adding a Transient Subscription

The only way to receive events on an already running object is to use transient subscription. Unlike persistent subscription, there is no administrative way to add a transient subscription. You have to program against the COM+ Catalog using the catalog objects and interfaces discussed in Chapter 6. In addition, it is your responsibility to remove the transient subscription from the Catalog when the subscribing component is released or if you want to unsubscribe.
Like a persistent subscriber, the object has to implement a sink interface for receiving events. The transient subscriber can choose to subscribe to all the sinks a particular event class supports, to a particular interface, or even to a particular method on a particular interface.
To add a transient subscription, you must follow these steps:

1. Create the catalog object (`CLSID_COMAdminCatalog`) and get a pointer to `ICOMAdminCatalog`.
2. Call `ICOMAdminCatalog::GetCollection( )` to retrieve a collection called `TransientSubscription` and get back an `ICatalogCollection` interface pointer.
3. Call `ICatalogCollection::Add( )` to get `ICatalogObject`.

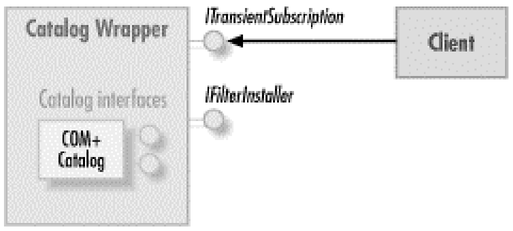
4. Call `ICatalogObject::put_Value( )` once for each desired property of the transient subscription you want to set. Some examples are the event class you want to subscribe to, subscribing interfaces, and the subscription name. An important property you need to set is whether or not you want to enable the subscription.
5. Call `ICatalogCollection::SaveChanges( )`.
6. Release everything.

You are required to perform a similar sequence to remove the transient subscription.
In fact, as you will see later on, managing a transient subscription is not the only feature of COM+ events that requires programming against the COM+ Catalog: implementing, adding, and removing a publisher filter and transient subscriptions filtering are also only available programmatically. In all these cases, the developer is required to program against the Catalog interfaces.
The Catalog interfaces have the following limitations:

- They are not type safe:
    - A BSTR is used for representing GUID, IID, and CLSID.
    - A BSTR is used instead of normal string.
    - Amorphous Variants are used to represent many data types.
- The COM+ interfaces and the underlying programming model and objects hierarchy require tons of generic code for iterating over the Catalog, even for simple tasks.
- The resulting code is tedious and error prone.

To alleviate the situation, I developed an easy-to-use wrapper object around the COM+ Catalog. The wrapper object saves you the agony of programming directly against the Catalog, reducing hundreds of lines of code to a mere line or two.
The wrapper object encapsulates the catalog objects and interfaces, exposing instead simple custom interfaces (with type safety) that perform all the hard work for you (see Figure 9-7). The wrapper object interfaces provide one-stop shopping for easy management of transient subscriptions and publisher filtering, providing you the same functionality as the Catalog interfaces with a fraction of the code.

Figure 9-7. The Catalog wrapper helper object



In the rest of this chapter, the use of the wrapper object will be demonstrated. Its implementation will also be described. The wrapper object source files are available on this book's web site, http://oreilly.com/catalog/comdotnetsvs/.

The first thing you will use the wrapper object for is registering a transient subscription with the COM+ Catalog. The Catalog wrapper encapsulates the code required to register a transient subscription by exposing the `ITransientSubscription` interface, defined as:

```
interface ITransientSubscription : IUnknown
{
   HRESULT Add([in,string]LPCWSTR pwzName,[in]CLSID
clsidEventClass,
               [in]REFIID iidInterface,[in]IUnknown
*pSink);
   HRESULT Remove([in,string]LPCWSTR pwzName);

   HRESULT AddFilter([in,string]LPCWSTR pwzSubName,
                     [in,string]LPCWSTR pwzCriteria);
   HRESULT RemoveFilter([in,string]LPCWSTR pwzSubName);
};
```

`ITransientSubscription` provides you with everything you need to easily manage a transient subscription—you can add a subscription to all the interfaces of a specified event class or to a particular interface on that class. Later, you will use `ITransientSubscription` to install or remove a transient subscriber-side filter.

Adding a transient subscription using the helper object is a one liner—a vastly simplified programming model that completely encapsulates the underlying Catalog. After initializing a pointer to a sink (`pMySink`) that you want to receive events on, create the wrapper object using `CLSID_CatalogWrapper` and call `TransientSubscription::Add( )`:

```
//Creating the wrapper object:
ITransientSubscription* pTransSubs = NULL;


::CoCreateInstance(CLSID_CatalogWrapper,...,IID_ITransien
tSubscription,
                   (void**)&pTransSubs);

//Adding a transient subscription:
pTransSubs ->Add(L"My Subs",CLSID_MyEventClass,
                  IID_NULL,//All interfaces of the event
class
                  pMySink);

//When you wish to unsubscribe:
pTransSubs ->Remove(L"My Subs");

//Releasing the wrapper object:
pTransSubs ->Release(  );
```

When you add a subscription, you provide the Catalog wrapper with the subscription name—a string identifying the subscription. The name is used to identify the subscription when you want to remove it later or when you want to associate a filter with it.
Transient subscriptions are more efficient than persistent subscriptions because they do not require you to pay the overhead of creating the object. However, transient subscriptions raise some lifetime issues of classic COM tightly-coupled events. Another deficiency of transient subscriptions is that the party adding them has to have administrative privileges to modify the Catalog.

9.5 Delivering Events

Once an event is published, COM+ is responsible for delivering the event to the subscribers. By default, publishers have very little to do with the delivery itself, to ensure decoupling of publishers from subscribers. However, COM+ does provide you ways to fine-tune the delivery and obtain additional information on the result of firing the event to the subscribers.

9.5.1 Serial Versus Parallel Publishing

Events by default are fired serially at subscribers—COM+ goes through the list of subscribers and publishes to them one at a time. The call to the event class does not return to the publisher until the last subscriber is notified. As a result, the publisher is blocked during publishing. To minimize the blocking time, you can configure your event class to use multiple threads for publishing by checking the "Fire in parallel" checkbox in the Advanced tab of the event class properties page (see Figure 9-4).
This setting is a mere request that COM+ will fire in parallel, and COM+ is not required to comply. COM+ uses threads from the RPC pool of threads to publish to subscribers, so parallel publishing is subjected to pool limitations. You should consider Fire in parallel as an optimization technique only; avoid relying on it in your design. For example, do not count on having all the subscribers get the event at the same time.

9.5.2 Error Handling

When an event class succeeds in publishing to all the subscribers, it returns `S_OK` to the publisher. If the event is delivered to COM+ but there are no subscribers, the return code is `EVENT_S_NOSUBSCRIBERS`. If the event is delivered, but is unable to invoke any of the subscribers, the return code is `EVENT_E_ALL_SUBSCRIBERS_FAILED`. In the case of partial delivery (an event that invokes some, but not all, subscribers), the return code is `EVENT_S_SOME_SUBSCRIBERS_FAILED`.
To promote loose coupling between the publisher and the subscribers, COM+ does not provide success or failure information about delivery for particular subscribers. The rationale is that publishers should not care about particular subscribers.
However, if your publisher does care about success or failure when delivering events to particular subscribers, you can implement a publisher filter to handle this case, which is discussed in the next section.

9.5.3 Publishing Order

COM+ does not, by default, provide a way to specify the order in which an event gets delivered to multiple subscribers. The publisher fires at the event class, and under-the-hood COM+ scans the list of subscribers and publishes to them. The events are published one at a time to the subscribers, in no determined or necessarily repeatable order. Publishers can control the order in which subscribers receive an event by implementing a publisher filter.

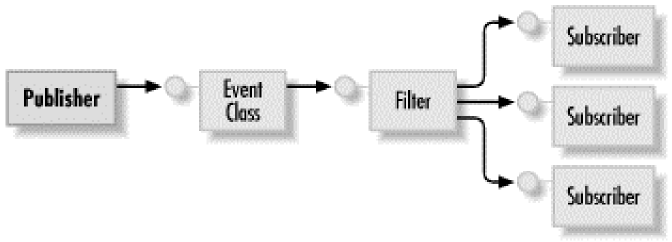
9.6 Event Filtering

If you would like to alter the default publish/subscribe behavior, COM+ provides a mechanism called *event filtering.* There are two kinds of filtering. The first, *publisher filtering*, lets you change the way events are published and therefore affect all the subscribers for an event class. The second, *subscriber filtering,* affects only the subscriber using that filter.
Both kinds of filters usually let you filter events without changing the publisher or the subscriber code. However, I find that event filtering is either cumbersome to use and implement, or limited and incomplete in what it offers. Those shortcomings are mitigated by the use of the COM+ Catalog wrapper object.

9.6.1 Publisher Filtering

Publisher filtering is a powerful mechanism that gives the publisher fine-grained control over event delivery. You can use a filter to publish to only certain subscribers, control the order in which subscribers get an event, and find out which subscribers did not get an event or had encountered an error processing it. The publisher-side filter intercepts the call the publisher makes to the event class, applies filtering logic on the call, and performs the actual publishing (see Figure 9-8).

Figure 9-8. A publisher filter



If you associate a filter with an event class, all events published using that class go through the filter first. You are responsible for implementing the filter (you will see how shortly) and to register it in the COM+ Catalog. The publisher filter CLSID is stored in the COM+ Catalog as a property of the event class that it filters. At any given time, an event class has at most one filter CLSID associated with it. As a result, installing a new filter overrides the existing one. When a publisher fires events on the event class, COM+ creates the publisher object and lets it perform the filtering.

9.6.1.1 Implementing a publisher filter

A publisher-side filter is a COM object that implements an interface called `IMultiInterfacePublisherFilter`. The filter need not necessarily be a COM+ configured component. The filter interface name contains the word Multi because it filters all the events fired on all the interfaces of the event class. Another interface, called `IPublisherFilter`, allows you to associate a filter with just one sink interface supported by an event class. It is still mentioned in the documentation, but has been deprecated (i.e., don't use it). The definition for `IMultiInterfacePublisherFilter` is:

```
interface IMultiInterfacePublisherFilter : IUnknown
{
   HRESULT Initialize([in]IMultiInterfaceEventControl*
                         pMultiInterfaceEventControl);

   HRESULT PrepareToFire([in]IID* piidSink,[in]BSTR
bstrMethodName,
                            [in]IFiringControl*
pFiringControl);
}
```

Only COM+ calls the methods of IMultiInterfacePublisherFilter as part of the event publishing sequence. If an event class has a publisher filter object associated with it, COM+ CoCreates the filter object and calls the Initialize( ) method when the publisher CoCreates the event class.
Each time the publisher fires an event at the event class, instead of publishing the event to the subscribers, COM+ calls the PrepareToFire( ) method and lets you do the filtering. When the publisher releases the event class, COM+ releases the filter object.
When the Initialize( ) method is called, COM+ passes in as a parameter an interface pointer of type IMultiInterfaceEventControl, defined as:

```
interface IMultiInterfaceEventControl : IUnknown
{
   HRESULT GetSubscriptions(
               [in] IID* piidSink,
               [in] BSTR bstrMethodName,
               [in] BSTR bstrCriteria,
               [in] int* nOptionalErrorIndex,
               [out, retval] IEventObjectCollection**
ppCollection);
 //Other methods

}
```

The only method of IMultiInterfaceEventControl relevant to publisher-side filtering is GetSubscriptions( ), used to get the list of subscribers at the time the event is published. Since COM+ calls the Initialize( ) method only once, you should cache the IMultiInterfaceEventControl pointer as a member variable of the filter object.
The actual filtering work is performed in the scope of the PrepareToFire( ) method. The first thing you need to do in the PrepareToFire( ) method is call the IMultiInterfaceEventControl::GetSubscriptions( ) method, passing an initial filtering criteria in as a parameter.
Filtering criteria are mere optimizations—a filter is used to inspect subscribers, and the filter may provide COM+ with an initial criterion of which subscribers to even consider for publishing.
The criterion is a BSTR containing some information about the subscribers. For example, consider a filtering criterion of the form:

```
_bstr_t bstrCriteria = "EventClassID == {F89859D1-6565-11D1-88C8-0080C7D771BF} AND
MethodName = \"OnNewOrder\"";
```

This causes COM+ to retrieve only subscribers that have subscribed to the specified event class and for the method called OnNewOrder on one of the event class interfaces.

Another example of a criterion is `ALL`, meaning just get all the subscribers. See the `IMultiInterfaceEventControl` documentation for more information on the exact criteria syntax.
`GetSubscriptions( )` returns an interface pointer of type `IEventObjectCollection`, which you use to access the subscribers collection.
Next, you call `IEventObjectCollection::get_NewEnum( )` to get an enumerator of type `IEnumEventObject` to iterate over the subscribers collection. While you iterate, you get one subscriber at a time in the form of `IEventSubscription`. You retrieve the `IEventSubscription` properties (such as the subscriber's name, description, IID), apply filtering logic, and decide if you want to publish to that subscriber. If you want to fire the event at that subscriber, use the last parameter passed to `PrepareToFire( )`, a pointer of type `IFiringControl`, passing in the Subscriber interface:

```
pFiringControl->FireSubscription(pSubscription);
```

At this point, you also get the exact success code of publishing to that particular subscriber. You then release the current subscriber and continue to iterate over the subscription collection.
If you want to publish to the subscribers in a different order than the one in which COM+ handed them to you, you should iterate over the entire collection, copy the subscribers to your own local list, sort the list to your liking, and then fire.

9.6.1.2 The CGenericFilter helper class

By now, you probably feel discouraged from implementing a publisher-side filter. The good news is that the filtering plumbing is generic, so I was able to implement all of it in an ATL COM object called `CGenericFilter` . `CGenericFilter` performs the messy interaction with the COM+ event system required of a publisher filter. All you have to do is provide the filtering logic (which is what a filter should do).
As part of the source files available with this book at O'Reilly's web site, you will find the Filter project—an ATL project containing the implementation of the `CGenericFilter` class. `CGenericFilter` lets you control which subscribers to publish to. If you want a different filter, such as one that controls the publishing order, you can implement that filter yourself, using the source files as a starting point.
The `CGenericFilter` class definition is (with some code omitted for clarity):

```
class CGenericFilter: public
CComObjectRootEx<CComSingleThreadModel>,
                      public
CComCoClass<CGenericFilter,&CLSID_MyFilter>,
                      public
IMultiInterfacePublisherFilter
{
   public:
   CGenericFilter(  );
   void FinalRelease(  );
   BEGIN_COM_MAP(CGenericFilter)
     COM_INTERFACE_ENTRY(IMultiInterfacePublisherFilter)
   END_COM_MAP(  )

   //IMultiInterfacePublisherFilter
   STDMETHOD(Initialize)(IMultiInterfaceEventControl*
pMultiEventControl);
   STDMETHOD(PrepareToFire)(IID* piidSink, BSTR
bstrMethodName,
                            IFiringControl*
pFiringControl);

   //Helper methods, used for domain logic specific
filtering
   HRESULT ExtractSubscriptionData(IEventSubscription*
pSubscription,
                                   SubscriptionData*
pSubscriptionData)const;
   BOOL ShouldFire(const SubscriptionData&
subscriptionData)const;
                  _bstr_t GetCriteria(  )const;

   IMultiInterfaceEventControl* m_pMultiEventControl;
};
```

The only thing you have to provide is the application domain-specific filtering logic, encapsulated in the two simple helper methods: `CGenericFilter::ShouldFire( )` and `CGenericFilter::GetCriteria( )`. The `CGenericFilter` implementation calls `GetCriteria( )` once per event to allow you to provide a filtering criteria. The default implementation returns `ALL`:

```
_bstr_t CGenericFilter::GetCriteria(  )const
{
   _bstr_t bstrCriteria = "ALL";//ALL means all the
subscribers,
                                //regardless of event
classes

   return bstrCriteria;
}
```

`CGenericFilter::ShouldFire( )` is the most interesting method here. `CGenericFilter` calls the method once per subscriber for a particular event. It passes in as a parameter a custom struct of type

SubscriptionData, which contains every available bit of information about the subscriber—including the name, description, and machine name:

```
struct SubscriptionData
{
   _bstr_t  bstrSubscriptionID;
   _bstr_t  bstrSubscriptionName;
   _bstr_t  bstrPublisherID;
   _bstr_t  bstrEventClassID;
   _bstr_t  bstrMethodName;
   _bstr_t  bstrOwnerSID;
   _bstr_t  bstrDescription;
   _bstr_t  bstrMachineName;
   BOOL     bPerUser;
   CLSID    clsidSubscriberCLSID;
   IID      iidSink;
   IID      iidInterfaceID;
};
```

ShouldFire( ) examines the subscriber and returns TRUE if you wish to publish to this subscriber or FALSE otherwise.

An example for implementing filtering logic in ShouldFire( ) is to publish only to subscribers whose description field in the Component Services Explorer says Paid Extra. See Example 9-2.

Example 9-2. Base your implementation of Shouldfire() on the information in SubscriptionData

```
BOOL CGenericFilter::ShouldFire(const SubscriptionData&
subscriptionData)const
{
   if(subscriptionData.bstrDescription == _bstr_t("Paid
Extra"))
      return TRUE;
   else
      return FALSE;
}
```

Finally, Example 9-3 shows the CGenericFilter implementation of PrepareToFire( ), which contains all the interaction with the COM+ event system outlined previously; some error-handling code was removed for clarity.

Example 9-3. CGenericFilter implementation of PrepareToFire( )

```
STDMETHODIMP CGenericFilter::PrepareToFire(IID* piidSink,
BSTR bstrMethodName,

IFiringControl* pFiringControl)
{
   HRESULT hres = S_OK;
   DWORD dwCount = 0;
   IEnumEventObject* pEnum = NULL;
   IEventSubscription* pSubscription = NULL;
   IEventObjectCollection* pEventCollection = NULL;

   _bstr_t bstrCriteria = GetCriteria(  );//You provide
the criteria

   hres = m_pMultiEventControl-
>GetSubscriptions(piidSink,

bstrMethodName,

bstrCriteria,NULL,

&pEventCollection);

   //Iterate over the subscribers, and filter in this
example by name
   hres = pEventCollection->get_NewEnum(&pEnum);
   pEventCollection->Release(  );

   while(TRUE)
   {
      hres = pEnum-
>Next(1,(IUnknown**)&pSubscription,&dwCount);
      if(S_OK != hres)
      {
         //Returns S_FALSE when no more items
         if(S_FALSE == hres)
         {
            hres = S_OK;
         }
         break;
      }
      long bEnabled = FALSE;
      hres = pSubscription->get_Enabled(&bEnabled);
      if(FAILED(hres) || bEnabled == FALSE)
      {
         pSubscription->Release(  );
         continue;
      }

      SubscriptionData subscriptionData;
      subscriptionData.iidSink = *piidSink;

      //A helper method for retrieving all of the
subscription
      //properties and packaging them in the handy
SubscriptionData
      hres =
ExtractSubscriptionData(pSubscription,&subscriptionData);
```

```
      if(FAILED(hres))
      {
         pSubscription->Release(  );
         continue;
      }
      //You provide the filtering logic in ShouldFire(  )
      BOOL bFire = ShouldFire(subscriptionData);
      if(bFire)
      {
         pFiringControl->FireSubscription(pSubscription);
      }
      pSubscription->Release(  );
   }
   pEnum->Release(  );

   return hres;
}
```

Again, let me emphasize that all you have to provide is the filtering logic in `ShouldFire( )` and `GetCriteria( )`; let `CGenericFilter` do the hard work for you.

9.6.1.3 Parameters-based publisher filtering

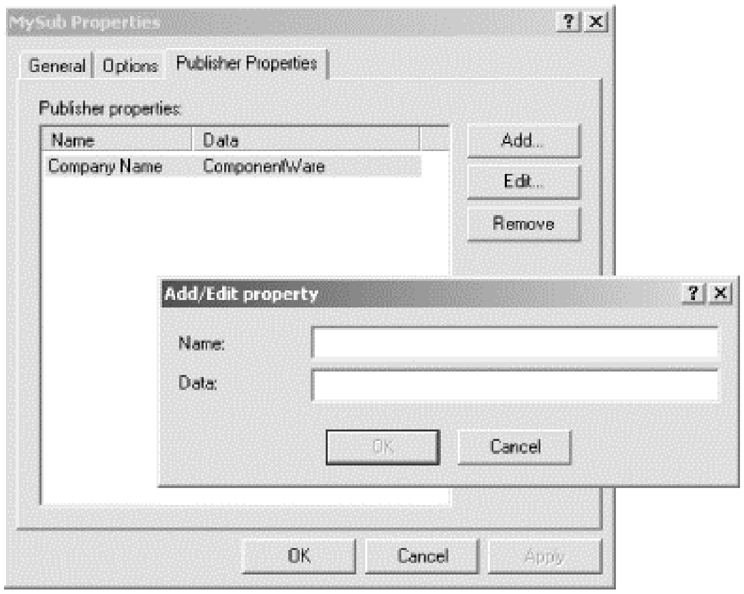
What begs an answer now (as I am sure you have already wondered) is why is `PrepareToFire( )` called "Prepare" if the event is fired there? Why not just call it `Fire( )`? It is called Prepare to support filtering based on the event parameters as well. In `PrepareToFire( )`, COM+ only tells you what event is fired.
What if you need to examine the actual event parameters to make a sound decision on whether or not you want to publish? In that case, the publisher filter can implement the same sink interfaces as the event class it is filtering.
After calling `PrepareToFire( )`, COM+ queries the filter object for the sink interface. If the filter supports the event interface, COM+ only fires to the filter. The filter should cache the information from `PrepareToFire( )` and perform the fine-tuned parameters-based filtering. In its implementation of the sink method, it uses `IFireControl` to fire the event to the client.

9.6.1.4 Custom subscription properties

Publisher-side filters usually base their filtering logic on the standard subscription properties—the subscription name, description, and so on. These properties are pre-defined and are available for every subscription. COM+ also allows you to define new custom properties for subscriptions and assign values to these properties, to be used by the publisher filter. Usually, you can take advantage of custom properties if you develop both the subscribing component and the publisher filter. You can define custom subscription properties administratively only for persistent subscribers.

To define a new custom property, display the subscription properties page, and select the Publisher Properties tab (the name is misleading). You can click the Add button to define a new property and specify its value (see Figure 9-9).

Figure 9-9. Defining new custom properties and assigning values on the Publisher Properties tab



Name	Data
Company Name	ComponentWare

Transient subscribers have to program against the component COM+ Catalog. Get hold of the transient subscription collection, find your transient subscription catalog object, and navigate from it to the `PublisherProperties` collection. You can then add or remove custom properties in the collection.

As explained before, when the publisher filter iterates over the subscription collection, it gets one subscriber at a time in the form of an `IEventSubscription` interface pointer. The filter can call `IEventSubscription::GetPublisherProperty( )`, specify the custom property name, and retrieve its value.

For example, here is how you retrieve a custom subscriber property called `Company Name`:

```
_bstr_t bstrPropertyName = "Company Name";
_variant_t varPropertyValue;
hres = pSubscription-
>GetPublisherProperty(bstrPropertyName,&varPropertyValue)
;
```

If the subscriber does not have this property defined, `GetPublisherProperty( )` returns `S_FALSE`. You can even define method parameter names as custom properties and specify a value or range in the property data. If the filter is doing parameters-based filtering, it can be written to parse the custom property value and to publish to that subscriber only when the parameter value is in that range.

9.6.1.5 Installing a publisher filter

There are two ways for associating a publisher filter with an event class. In the absence of any names for these two ways from the COM+ team at Microsoft, I call the first *static association* and the second *dynamic association*.

Static association requires you to program against the COM+ Catalog and store the filter CLSID as a property of the event class. The filter will stay there until you remove it or override it with another CLSID. Static association affects all publishers that use that event class, in addition to all instances of the event class.

Dynamic association takes place at runtime. The publisher will not only create an event class, but also directly creates a filter object and associates it only with the instance of the event class it currently has. Dynamic association affects only the publishers that use that particular instance of the event class. Dynamic association does not persist beyond the lifetime of the event class object. Once you release the event class, the association is gone. Dynamic association allows a publisher to bind a particular instance of an event class with a particular instance of a filter; it overrides any static filter currently installed.

The main disadvantage of dynamic association is that you cannot dynamically associate a filter with an instance of a queued event class (discussed later on), since you are interacting with the recorder for the event class, not the event class itself.

9.6.1.6 Static association of a publisher filter with an event class

To statically associate a publisher filter CLSID with the event class you want it to filter, you have to follow these steps:

1. Create the catalog object.
2. Get the `Applications` collection.
3. For each application in the collection, get the `Components` collection.
4. Iterate through the `Components` collection looking for the event class. If the class is not found, get the next `Application` collection and scan its `Components` collection.
5. Once you find the event class, set the `MultiInterfacePublisherFilterCLSID` event class property to the CLSID of the filter.
6. Save changes on the `Components` collection and release everything.

Again, the Catalog wrapper helper object is useful, as it saves you the interaction with the COM+ Catalog. The helper object implements an interface called `IFilterInstaller`, defined as:

```
interface IFilterInstaller : IUnknown
{
  HRESULT Install([in]CLSID clsidEventClass,[in]CLSID
clsidFilter);
  HRESULT Remove ([in]CLSID clsidEventClass);
};
```

`IFilterInstaller` makes adding a filter a breeze—just specify the CLSID of the event class and the CLSID of the filter, and it will do the rest for you:

```
HRESULT hres = S_OK;

hres =
::CoCreateInstance(CLSID_CatalogWrapper,NULL,CLSCTX_ALL,

IID_IFilterInstaller,(void**)&pFilterInstaller);

hres = pFilterInstaller-
>Install(CLSID_MyEventClass,CLSID_MyFilter);

pFilterInstaller->Release( );
```

Note that you do not need to specify the application name as a parameter; just specify the event class and the filter CLSID. Use `IFilterInstaller::Remove( )` to remove any filter associated with a specified event class.

9.6.1.7 Dynamic association of a publisher filter with an event class

To associate a publisher filter object with an event class instance dynamically, follow these steps:

1. Create the event class and get the sink interface.
2. Query the event class for `IMultiInterfaceEventControl` interface.
3. Create the filter object.
4. Call `IMultiInterfaceEventControl::SetMultiInterfacePublisherFilter( )` and pass in the filter object.
5. Release `IMultiInterfaceEventControl`.
6. Publish events to the event class object. The events will go through the filter you have just set up.
7. Release the event class and the filter when you are done publishing.

Example 9-4 shows some sample code that uses this technique.

Example 9-4. Installing a publisher-side filter dynamically

```
HRESULT hres = S_OK;
IMySink* pMySink = NULL;
IMultiInterfacePublisherFilter* pFilter = NULL;
IMultiInterfaceEventControl* pEventControl = NULL;

//Create the filter
hres = ::CoCreateInstance(CLSID_MyFilter,NULL,CLSCTX_ALL,

IID_IMultiInterfacePublisherFilter,(void**)&pFilter);
//Create the event class
hres =
::CoCreateInstance(CLSID_MyEventClass,NULL,CLSCTX_ALL,
                          IID_IMySink,(void**)&pMySink);

//Query the event class for IMultiInterfaceEventControl
hres = pMySink -
>QueryInterface(IID_IMultiInterfaceEventControl,
                               (void**)pEventControl);

//Setting the filter
hres = pEventControl-
>SetMultiInterfacePublisherFilter(pFilter);
pEventControl->Release(  );

//Firing the event
hres = pMySink->OnEvent1(  );//The event is now filtered

pMySink->Release(  );
pFilter->Release(  );
```

Unfortunately, COM+ has a bug regarding correct handling of dynamically associating a publisher filter with an event class. COM+ does not call the filter method `IMultiInterfacePublisherFilter::Initialize( )`, and as a result, you can't do much filtering. I hope this situation will be fixed in a future release of COM+.

This defect, plus dynamic association's inability to work with queued event classes, renders it effectively useless. Avoid dynamic association of a publisher filter; use static association instead.

9.6.2 Subscriber-Side Filtering

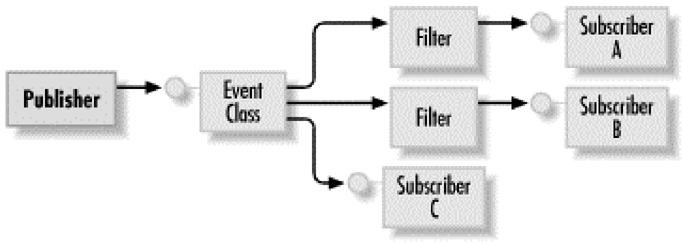
Not all subscribers have meaningful operations to do as a response to every published event. Your subscriber may want to take action only if your favorite stock is trading, or perhaps only if it is trading above a certain mark. One possible course of action is to accept the event, examine the parameters and decide whether to process the event or discard it.

However, this action is inefficient if the subscriber is not interested in the event for the following reasons:

- It forces a context switch to allow the subscriber to examine the event.
- It adds redundant network round trips.
- Writing extra examination code may introduce defects and require additional testing.
- Event examination and processing policies change over time and between customers. You will chase your tail trying to satisfy everybody.

What you should really do is to put the filtering logic outside the scope of the subscriber. You should have an administrative, configurable, post-compilation, deployment-specific filtering ability. This is exactly what subscriber-side filtering is all about (see Figure 9-10). Subscribers that do not want to be notified of every event published to them, but want to be notified only if an event meets certain criteria, can specify filtering criteria.

Figure 9-10. Specifying filtering criteria for a persistent subscriber



A subscriber-side filter is a string containing the filtering criteria. For example, suppose you subscribe to an event notifying you of a new user in your portfolio management system, and the method signature is:

```
HRESULT OnNewUser([in]BSTR bstrName,[in]BSTR bstrStatus);
```

You can specify such filtering criteria as:

```
bstrName = "Bill Gates" AND bstrStatus = "Rich"
```

The event will only be delivered to your object if the username is Bill Gates and his current status is Rich.

The filter criteria string recognizes relational operators for checking equality (`==`,`!=`), nested parentheses, and logical keywords `AND`, `OR`, and `NOT`. COM+ evaluates the expression and allows the call through only if the criteria are evaluated to be true.

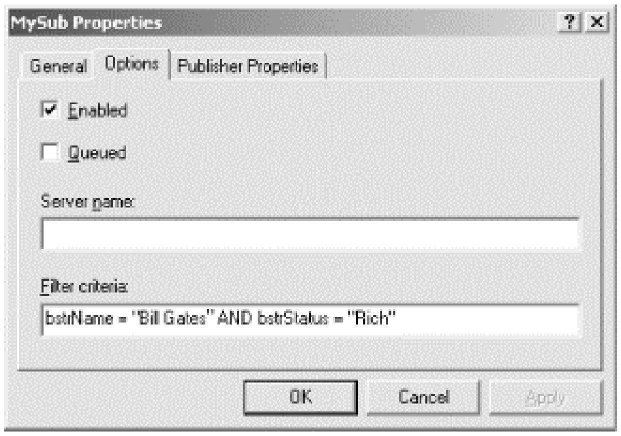
If you have wrong parameters or spelling mistakes, or if the parameter names were changed, the subscriber will never be notified.

Because subscriber-side filtering occurs only after the event has been fired, if a publisher filter is used, then the event has to pass the publisher filter first. The obvious conclusion is that publisher-side filtering takes precedence over subscriber-side filtering.

9.6.2.1 Persistent subscriber-side filtering

Only persistent subscribers can specify a subscriber filter administratively. They can do so by displaying the persistent subscription properties page, selecting the Options tab, and specifying the Filter criteria (see Figure 9-11).

Figure 9-11. Subscriber-side filtering



☑ Enabled

☐ Queued

Server name:

Filter criteria:

bstrName = "Bill Gates" AND bstrStatus = "Rich"

9.6.2.2 Transient subscriber-side filtering

Transient subscribers have to program against the Catalog to set a transient subscription filter criteria, following similar steps to those performed when registering a transient subscription:

1. Get hold of the Catalog interface.
2. Get the transient subscription collection object.
3. Find your transient subscription.
4. Set a subscription property called `FilterCriteria` to the string value of your filter.
5. Save changes and release everything.

The Catalog wrapper's interface `ITransientSubscription`, discussed earlier, allows you to add (or remove) a subscriber-side filter to a transient subscription with the `AddFilter( )` and `RemoveFilter( )` methods. The methods accept the subscription name and a filtering string.
Example 9-5 demonstrates the same example from the persistent subscriber filter, but for a transient subscriber for the same event.

Example 9-5. Adding a transient subscription filtering criteria using the wrapper object

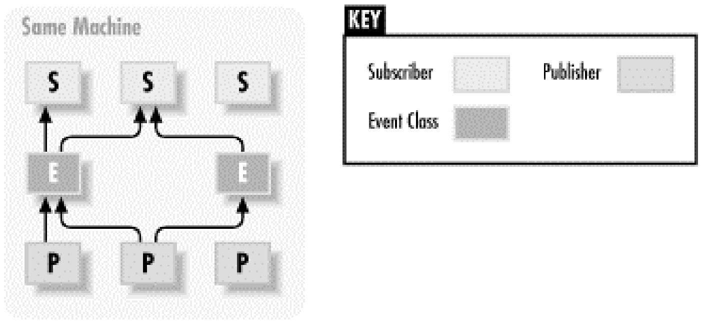
```
//Adding a transient subscription filter:
LPCWSTR pwzCriteria = L"bstrUser = \"Bill Gates\" AND
bstrStatus = \"Rich\""

//"MySubs" is the transient subscription name

hres = pTransSubs->AddFilter(L"MySubs",pwzCriteria);


//Or removing the filter:
pTransSubs ->RemoveFilter(L"MySubs");
```

The main disadvantage of a transient subscriber filter compared to a persistent subscriber filter is that you hardcode a filter, which is sometimes deployment- or customer-specific. Persistent subscribers can always change the filtering criteria using the Component Services Explorer during deployment.

9.7 Distributed COM+ Events

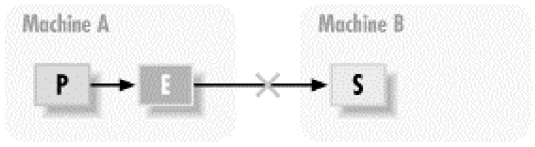
As long as the publisher, the event class, and the subscribers are all installed on the same machine, you can have pretty much any topology of interaction (see Figure 9-12). On the same machine, publishers can publish to any event class, event classes can deliver events to any subscriber, and subscribers can subscribe to as many event classes as they like.

Figure 9-12. You can have any publisher and subscriber topology on the same machine



Unfortunately, the COM+ event service has a serious limitation—the event class and all its subscribers have to be on the same machine. This means that a deployment, such as the one shown in Figure 9-13, is not possible.

Figure 9-13. The event class and the subscriber must reside on the same machine

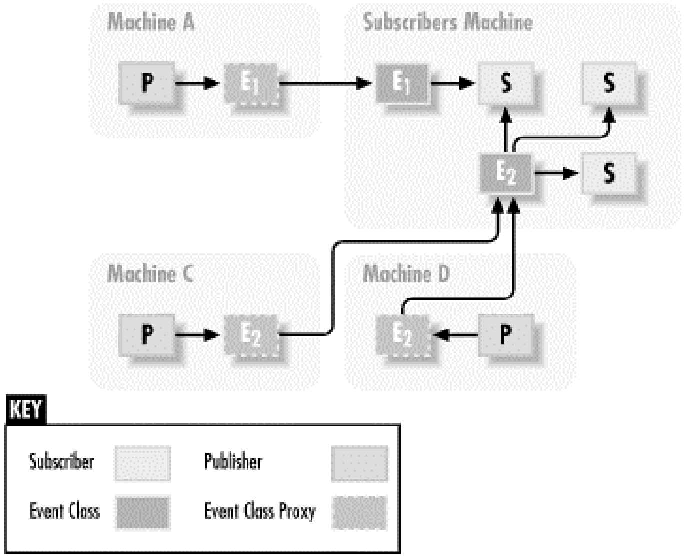


The rest of this section presents you with a few workaround solutions for this problem that allow you to distribute your events across the network. All the solutions adhere to the limitation that the event class and the subscribers have to reside on the same machine, and they solve the problem by designing around it. Like most things in life, each solution has pros and cons. It will be ultimately up to you, the system designer, to select the most appropriate solution for your domain problem at hand.

9.7.1 Solution 1: One Machine for All the Subscribers and Event Classes

This solution is the simplest to implement. You install all event classes on one machine, along with all subscribers. You install the event classes in a COM+ server application and generate a proxy installation (see Chapter 1) for the event classes' application. (Remember, the event class application has to be a server application for you to export it as a proxy application.) You then deploy the event class proxy application on all the machines that host publishers, making sure the proxy applications point to the event classes/subscribers machine (see Figure 9-14). .

Figure 9-14. This solution requires having all subscribers and event classes on the same machine



When a publisher on a remote Machine A wants to fire an event of type E_1, it creates a proxy for that event class and calls the event method on it. The event call will be marshaled to the place where the event class resides—on the subscribers machine—and get published to all the subscribers that subscribed to it. It is also very easy for subscribers to subscribe to more than one event class, since all the event classes are installed locally on the subscribers machine

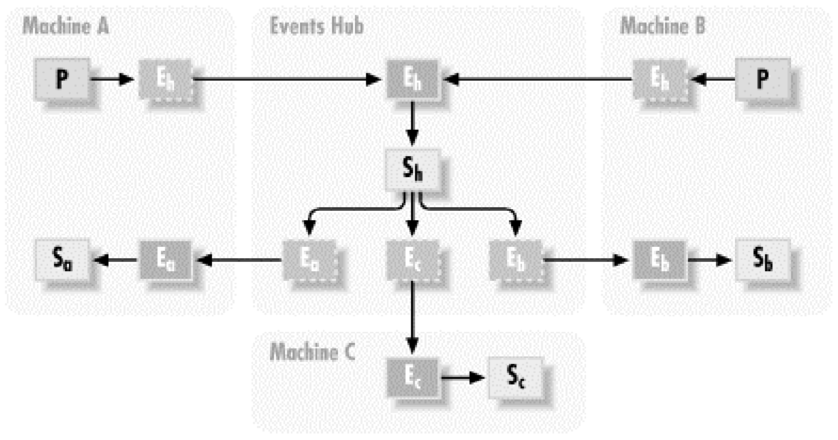
This solution has the following disadvantages:

1. By locating the event classes away from the publishers, you introduce extra expensive round trips across the network.
2. The single machine hosting all the event classes and the subscribers becomes a hot spot for performance. The machine CPU and operating system have to handle all the traffic. There is no load balancing in your product, and load balancing is a major reason for distributing your components in the first place.
3. The subscribers machine solution is a single point of failure in your system.
4. The subscribers are not necessarily ideally deployed. If the subscribers do not have to reside where the event classes are, you may have put them somewhere else—maybe on the same machine where the database is if they have to access it frequently. Performance may suffer.

9.7.2 Solution 2: Machine-Specific Event Classes

This solution allows you to distribute your subscribers anywhere, according to whatever design preference you have. This distribution makes it possible for you to publish from one machine to subscribers that reside on multiple other machines (see Figure 9-15). However, this particular solution is more complex to manage and deploy than the previous solution.

Figure 9-15. A hub machine has machine-specific event class proxies used to distribute events



The idea behind this solution is to create a COM+ events hub on one designated machine. The hub machine is responsible for distributing the events to where the subscribers really reside.
This solution uses two kinds of event classes. The first is an event class that resides only on the hub machine, called E_h. You install proxies to E_h on all the publishers' machines. Publishers will only publish using E_h.
The second kind of event class is a machine-specific event class. Every machine that hosts subscribers has its own dedicated event class type, installed only on that machine. In Figure 9-15, these types are E_a, E_b, and E_c, corresponding to the three machines in the figure. You need to install a proxy to every machine-specific event classes on the hub machine. All event classes in this solution support the exact same set of sink interfaces.
When a publisher on Machine A wants to publish an event to subscribers on Machines A, B, and C, the publisher on Machine A creates an instance of the E_h event class (which only actually creates a proxy) and fires to it. The E_h proxy forwards the call to where E_h really executes—on the hub machine. On the hub machine there is a hub subscriber (S_h) that subscribes to the E_h event. The way S_h handles the event is to create all the machine-specific event classes (E_a, E_b, and E_c) and fire that particular event to them. Because there are only proxy installations of the machine-specific event classes on the hub machine, the event is distributed to multiple machines, where local subscribers—the real subscribers—handle the event.
The main advantage of using this solution is that it gives you complete freedom in locating your subscribers. However, the flexibility comes with a hefty price:

- When you publish, you encounter many expensive round trips across the network. Even if all the subscribers are on the publisher machine, the publisher still has to go through the hub machine.
- You have to duplicate this solution for every kind of event class you have, and you therefore end up with separate sets of machine-specific and hub event classes.
- The added complexity of this solution means that you probably have a deployment, administration, and maintenance nightmare on your hands.
- The hub machine is potentially a single point of failure in your system.

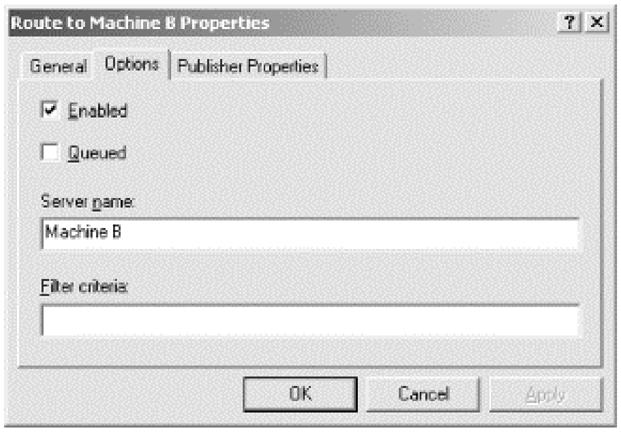
9.7.3 Solution 3: COM+ Routing

This last solution for distributing events to subscribers on multiple remote machines takes advantage of a feature provided for you by

COM+. However, it is a partial solution because it only works with persistent subscribers. If your application uses transient subscribers (as it most likely will), you have to use one of the two solutions discussed previously. The idea here is similar to the hub machine solution, and to distinguish between them, I call this one the *routing* solution.

COM+ provides a field called Server name on the Options tab for every persistent subscription (see Figure 9-16).

Figure 9-16. Instructing COM+ to create the subscriber object on the machine specified in the Server name field



☑ Enabled

☐ Queued

Server name: Machine B

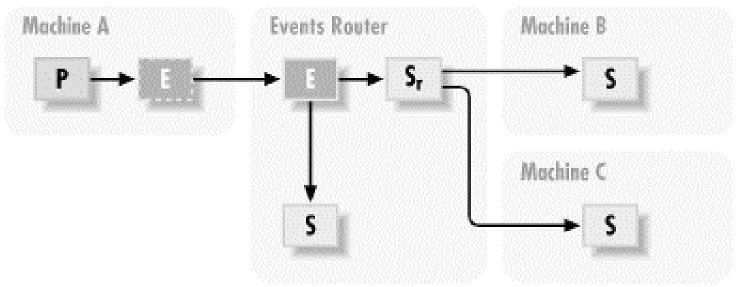
Filter criteria:

Whenever an event is published to a persistent subscriber, before CoCreating the subscriber object, COM+ first checks the value of the Server name property. If it is not an empty string, COM+ CoCreates the subscriber on the specified machine, fires the event to the sink interface, and releases the subscriber.

Routing events to multiple machines takes advantage of this feature. Instead of using machine-specific event classes like in Solution 2, the routing solution uses machine-specific persistent subscriptions.

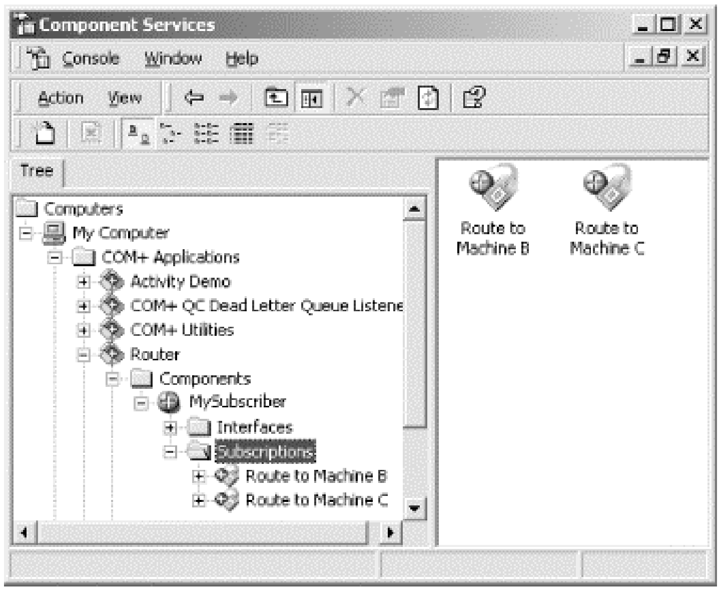
For example, suppose you have a publisher on Machine A and a subscribing component called MySubscriber that you want to deploy on Machines B and C. The publisher publishes using an event class called E. On Machines B and C you add subscriptions to the event class, to the locally installed copies of MySubscriber. You then install the MySubscriber component on another designated routing machine, together with the event class E, and install on Machine A only the proxy to E (see Figure 9-17).

Figure 9-17. The routing solution uses machine-specific subscriptions and a routing machine



To the installation of MySubscriber on the router machine (called S_R in Figure 9-17) add machine-specific subscriptions: for every deployment of MySubscriber on another machine, add a subscription and redirect the invocation to that machine, using the Server name field. See Figure 9-18.

Figure 9-18. The router machine has a machine-specific subscription used to route the event to corresponding machines



Now, when the publisher on Machine A CoCreates a proxy to the event class and fires an event at it, the call goes to the router machine. COM+ inspects the subscriptions on the router machine for the event class, detects the Server name in the subscriptions, creates the subscribers on the remote machines, and publishes to them.

I already pointed out the main drawback of this solution (persistent subscribers only), but it has a few others:

- Setting up and configuring the system is nontrivial effort. You have to either write some installation scripts to help you automate it or manually configure it at every customer deployment. Every customer site has its own machine names; you will not be able to specify the machine names in your application MSI file, exported for release.

- You have to go through the router machine, so you end up paying for an extra network hop.
- The router machine can be a performance bottleneck.
- The router machine is potentially a single point of failure.

9.8 Asynchronous Events

So far, in this discussion of the COM+ event model, it was always assumed that publishing the event is a synchronous operation—during publishing, the publisher is blocked and that blocking time is proportional to the number of subscribers and their individual processing times. A true loosely coupled event mechanism decouples the publisher from the subscriber even further. It allows the publishers to publish asynchronously and permits the subscribers to handle the event asynchronously as well.
COM+ provides this capability by using COM+ queued components (see Chapter 8). As you will see, both the event class and the subscribers can be queued components, to enable asynchronous publishing and subscribing.

9.8.1 Asynchronous Publishing

COM+ has a built-in service for asynchronous execution: queued components. COM+ events and queued components go together very well, giving you the benefits of a loosely coupled system and the flexibility of asynchronous execution.
Every event class supports a set of sink interfaces. As with any other COM+ component, you can configure any one of the sink interfaces as Queued. A publisher creates a queued event class using the queue moniker. When a publisher fires an event to a queued event class interface, COM+ performs its usual handling of a queued component (recording the call, placing it in a queue, and so on). The COM+ queued component listener pulls the messages (the events) from the event class queue and plays them back to the event class.
The publisher is blocked only for the relatively short period time it takes COM+ to record the call. Contrast this with the `Fire in Parallel` attribute, which returns control to the publisher only after all subscribers have been notified.
A publisher that is interested in creating a queued event class creates it using the `queue` moniker, like any other queued component. Because of the inherited limitation of queued components—that a queued component cannot reside in a library application—an event class that uses a queued component cannot be in a library application.
One interesting side effect of using queued components is that if you publish events on two queued event classes, events may not

replay in the order in which they were originally fired. This situation can be a source of trouble if the two publishing sessions are related in some manner. If having one event take place before another is important, you need to make the calls on the same specific queued event class.

9.8.2 Asynchronous Subscribers

COM+ can use queued components to invoke calls on a component that also uses persistent subscriptions. Because COM+ is the one that creates the subscriber, you have to let COM+ know that it should create the component using the `queue` moniker, rather than `CoCreateInstance( )`. You do that by checking the Queued property of the persistent subscription (see Figure 9-6).
When COM+ publishes to a queued subscriber, it posts a message to the subscriber's message queue. The listener of the COM+ application that hosts the subscriber will detect the messages in the queue, create a player, and play back the events to the subscriber. There are two main advantages of using queued subscribers:

- The publisher code remains the same for queued and normal subscribers, and it allows for lengthy processing of the event on the subscriber side, instead of having to spin off a worker thread, as if you were using classic COM.
- Having both the publisher and the subscriber using queued components allows both to work offline at the same time and be completely disconnected.

There are also two main disadvantages:

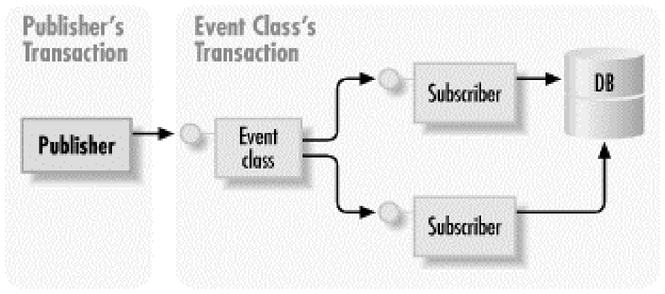
- The publisher is still blocked while looking through the subscribers list and, for each subscriber, while creating a recorder, posting messages to queues, and performing other queued component management activities.
- If somebody adds a nonqueued subscription to your system, then publishing is not fully asynchronous. The publisher is blocked while the nonqueued subscriber processes the event.

9.9 COM+ Events and Transactions

COM+ transactions flow downward from the transaction root, as you have seen in Chapter 4. New objects created during the transaction take part in their creator's transaction or are placed in a transaction of their own, according to their transaction configuration.

If the publisher takes part in a transaction, it is recommended that the subscribers participate in the publisher's transaction. But how would the transaction be propagated by the publisher to the subscriber if the publisher does not create the subscriber directly? To propagate the publisher's transaction to the subscriber, you should configure the event class to support or require transactions. Like any other COM+ component, the event class has a Transaction tab that applies to the COM+ synthesized implementation.
Adding the event class to your transaction will not affect the transaction voting result; in any COM+ context the consistency bit is set by default to `TRUE` and the COM+-provided implementation of the event class does not change that bit. You also need to configure the (persistent) subscriber component to support transactions. Now the subscriber takes part in the publisher's transaction and it can abort the publisher's transaction or vote to commit it.
There is one more thing you should keep in mind when mixing COM+ events and transactions: Do not configure the event class to require a new transaction. This causes the subscriber to take part in a separate transaction, the one initiated by the event class (see Figure 9-19).

Figure 9-19. Configuring the event class to require new transaction results in a separate transaction for the persistent subscriber

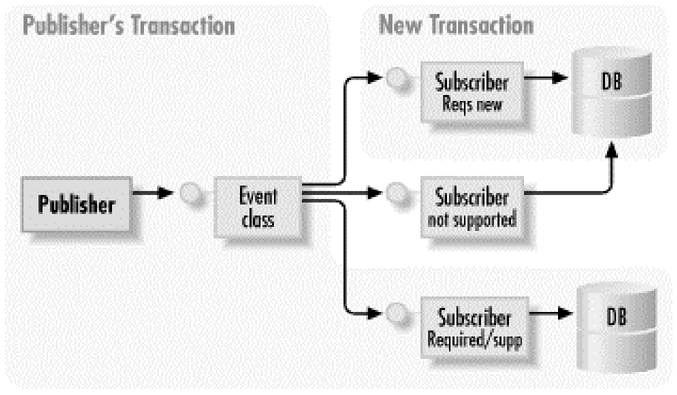


If the publisher's transaction is aborted, the subscriber's transaction can still commit successfully, which may involve changes to the database and other persistent changes to your system state. Nobody tells the subscriber to roll back those changes, despite the fact that the event that triggered the changes is fired from a transaction that aborts.
In addition, when the publisher tries again, the event may be fired once more, leaving the subscriber in an inconsistent state.

9.9.1 Persistent Subscribers and Transactions

Similarly, avoid configuring any persistent subscriber's transaction setting to Requires New and do not mix nontransactional subscribers with transactional ones; such practices may introduce unwelcome side effects when the publisher's transaction is aborted (see Figure 9-20).

Figure 9-20. Avoid configuring subscribers to require new transactions or to mix nontransactional subscribers with transactional ones



9.9.2 Transient Subscribers and Transactions

Transient subscribers are already instantiated when the event is fired and may be part of their creator's transaction. I can only recommend being mindful when combining transient subscriptions with a transactional publisher because you may end up with the same inconsistencies mentioned in the previous section.
In particular, transient subscribers should not abort their client/creator's transaction as a response to a publisher's event, an event that may have been fired from within another transaction. The problems that arise when you combine transient subscribers and publisher transactions are typical of passing object references across transaction boundarie. The object does not know whether it is allowed to abort the transaction or not (as discussed in Chapter 4).

9.10 COM+ Events and Security

The fact that the publisher does not call methods on the subscribers directly is an important software engineering capability. Nevertheless, you should never decouple your components at the expense of security. COM+ must still allow the system administrators to configure the access rights to subscribers. COM+ events take advantage of the rich security infrastructure offered by COM+, and COM+ also provides you with event system-specific security settings.

9.10.1 The Event Class and Role-Based Security

Like other configured components, an event class can use role-based security. The most common use of use role-based security for event classes is to control which publisher is allowed to fire events.

However, since roles in COM+ are per application, be sure to add roles and users for each product to the event class application if you intend to share event classes between a few applications and products.
You can use role-based security in another way: to implement a publisher-side filter that calls `ISecurityCallContext::IsCallerInRole( )` (discussed in Chapter 7) and controls the order of publishing based on the publisher's role.

9.10.2 Subscribers and Role-Based Security

The subscriber can use role-based security to control access to its services. Unlike an event class usage of role-based security (which affects the publishing side and therefore all the subscribers), when a subscriber uses role-based security, only that subscriber is affected by the access checks. If all your subscribers have uniform security requirements, putting the security check on the event class is the right decision because it improves performance (the publisher does not publish at all if it is not allowed to). However, if the security requirements of your subscribers vary (if some require tighter security than others), putting the security access checks on the sensitive subscribers may provide you with the better solution.

9.10.3 In-Process Subscribers

From a security point of view, an interesting situation arises when the event class and the subscriber component are both library applications. As a result, when the publisher CoCreates the event class and publishes to it, the subscriber is loaded into the publisher process.
Unlike a conventional library application that is intended to share the address space of its client (and may very well be developed by the same team), the publisher/subscriber relationship is much less trusting and coupled.
Most software vendors would feel uneasy letting an unknown entity into their process. The subscriber may be of dubious quality (and may take the publisher down with it when it crashes) or even malicious (I will leave it to your imagination what you can do if somebody lets you into their process).
To protect the publisher, the system administrator can enforce all subscribers to be created in their own process. On the Advanced tab of the Event Class properties page, if "Allow in-process subscribers" is not checked, the subscriber object will be created in a separate process, even if it is configured to run as a library application (see Figure 9-4).

9.10.4 Per-User Subscriptions

COM+ allows you to deliver an event to a particular subscriber only if a specific user is logged on to the publisher's machine. When the user logs off, the subscription is disabled. Per-user subscription requires the publisher and subscriber to be on the same computer, since logon and logoff are only detected locally in Windows.
To activate per-user Subscription you must set the `PerUser` flag on the subscription record to `TRUE` and specify a username. You can do that by programming against the COM+ Catalog.
Per-user subscription is an esoteric security mechanism, and I recommend using role-based security instead to achieve similar capabilities with a fraction of the code and restrictions.

9.11 COM+ Events Limitation

COM+ Events is an outstanding service that saves you a lot of work—it provides an extensible, feature-rich service that allows you to focus on adding value to your product, not on event connectivity plumbing.
However, the event system has a few limitations, and this chapter would not be complete without pointing them out. Knowing about them allows you to make the best use of COM+ events:

- As you have seen, COM+ events do not provide you with absolute location transparency. You have to jump through hoops to distribute your events across the enterprise.
- Good support for a very large number of subscribers (more than a few hundred) is lacking. To publish an event, COM+ maintains a linked list of subscribers, and it scans it on every event—i.e., publishing overhead is linear to a number of subscribers. There is no way to perform a broadcast.
- All parties involved (publisher, event class, and subscribers) have to run on a Windows 2000 machine. This is usually not a problem at the middle tier, but it does rule out most of the portable devices, such as laptops, PDAs, and cell phones.
- COM+ has difficulty handling a very large amount of data as parameters for events. Avoid large strings and huge arrays.
- COM+ events cannot handle a high rate of event publishing because it takes time to publish an event. If events are published faster than COM+ can handle them, you get memory bloating and COM+ will occasionally fail. On a stress test I conducted on COM+ events, I had three publishers, each on its own machine, creating an event class proxy and firing every 300 milliseconds at one subscriber on a fourth machine. COM+ failed after a day and a half.

9.12 Summary

COM+ loosely coupled events demonstrate all of the core COM+ component services principles discussed in this book: the service has evolved to improve an existing solution; it offers a spectrum of features, from simple, to administrative Component Services Explorer support, to advanced programmatic features; and it interacts with almost all of the other COM+ services, such as transactions, security, and queued components. Although this chapter has discussed the main points of the service, numerous other possibilities exist, including pooled persistent subscribers. The important lesson is that once you understand how each individual service works, you can start combining the services in powerful and synergetic ways. COM+ loosely coupled events are the last COM+ component service described in this book. You will now learn about .NET and see how it utilizes COM+ component services.

Chapter 10. .NET Serviced Components

.NET is the new platform from Microsoft used to build component-based applications, from standalone desktop applications to web-based applications and services. The platform will be available on forthcoming Microsoft operating systems and supported by the next release of Visual Studio, called *Visual Studio.NET*. In addition to providing a modern object-oriented framework for building distributed applications, .NET also provides several specialized application frameworks. These frameworks include Windows Forms for rich Windows clients, ADO.NET for data access, and ASP.NET for dynamic web applications. Another important framework is Web Services, which is used to expose and consume remote objects using the emerging SOAP and other XML-based protocols.

.NET is Microsoft's next-generation component technology. It is designed from the ground up to simplify component development and deployment, as well as to support interoperability between programming languages.

Despite its innovations and modern design, .NET is essentially a *component technology*. Like COM, .NET provides you with the means to rapidly build binary components, and Microsoft intends for .NET to eventually succeed COM. Like COM, .NET does not provide its own component services. Instead, .NET relies on COM+ to provide it with instance management, transactions, activity-based synchronization, granular role-based security, disconnected asynchronous queued components, and loosely coupled events. The .NET namespace that contains the types necessary to use COM+ services was named *System.EnterpriseServices* to reflect the pivotal role it plays in building .NET enterprise applications.

A .NET component that uses COM+ services is called a *serviced component* to distinguish it from the standard managed components in .NET. If you are not familiar with .NET, you should first read Appendix C or pick up a copy of *.NET Framework Essentials* by Thuan Thai and Hoang Lam (O'Reilly, 2001).

If you are already familiar with the basic .NET concepts, such as the runtime, assemblies, garbage collection, and C# (pronounced "C sharp"), continue reading. This chapter shows you how to create .NET serviced components that can take advantage of the COM+ component services that you have learned to apply throughout this book.

10.1 Developing Serviced Components

A .NET component that takes advantage of COM+ services needs to derive from the .NET base class `ServicedComponent`. `ServicedComponent` is defined in the `System.EnterpriseServices` namespace. Example 10-1 demonstrates how to write a .NET serviced component that implements the `IMessage` interface and displays a message box with "Hello" in it when the interface's `ShowMessage( )` method is called.

Example 10-1. A simple .NET serviced component

```
namespace MyNamespace

{
   using System.EnterpriseServices;
   using System.Windows.Forms;//for the MessageBox class


   public interface IMessage
   {
      void ShowMessage(  );
   }
   /// <summary>
   ///    Plain vanilla .NET serviced component
   /// </summary>
   public class MyComponent:ServicedComponent,IMessage
   {
      public MyComponent(  ) {}//constructor
      public void ShowMessage(  )
      {
        MessageBox.Show("Hello!","MyComponent");
      }
   }
}
```

A serviced component is not allowed to have parameterized constructors. If you require such parameters, you can either design around them by introducing a `Create( )` method that accepts parameters, or use a constructor string.

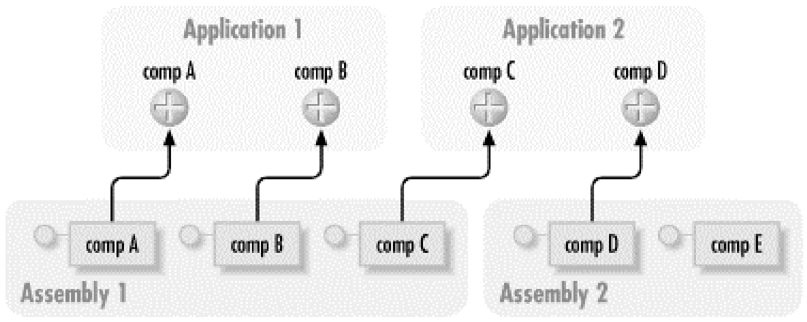
There are two ways to configure a serviced component to use COM+ services. The first is COM-like: you derive from `ServicedComponent`, add the component to a COM+ application, and configure it there. The second way is to apply special attributes to the component, configuring it at the source-code level. When the component is added to a COM+ application, it is configured according to the values of those attributes. Attributes are discussed in greater detail throughout this chapter as you learn about configuring .NET components to take advantage of the various COM+ services.

.NET allows you to apply attributes to your serviced components with great flexibility. If you do not apply your own attributes, a serviced component is configured using default COM+ settings when it is added to a COM+ application. You can apply as many attributes as you like. A few COM+ services can only be configured via the Component Services Explorer. These services are mostly deployment-specific configurations, such as persistent subscriptions to COM+ Events and allocation of users to roles. In general, almost everything you can do with the Component Services Explorer can be done with attributes. I recommend that you put as many design-level attributes as possible (such as transaction support or synchronization) in the code and use the Component Services Explorer to configure deployment-specific details.

10.2 .NET Assemblies and COM+ Applications

When you wish to take advantage of COM+ component services, you must map the assembly containing your serviced components to a COM+ application. That COM+ application then contains your serviced components, just like any other component—COM+ does not care whether the component it provides services to is a managed .NET serviced component or a classic COM, unmanaged, configured component. A COM+ application can contain components from multiple assemblies, and an assembly can contribute components to more than one application, as shown in Figure 10-1. Compare Figure 10-1 to Figure 1-8. There is an additional level of indirection in .NET because an assembly can contain multiple modules.

Figure 10-1. COM+ applications and assemblies



However, setting up an assembly to contribute components to more than one COM+ application is not straightforward and is susceptible to future registrations of the assembly. As a rule, avoid mapping an assembly to more than one COM+ application.

10.3 Registering Assemblies

To add the serviced components in your assembly to a COM+ application, you need to register that assembly with COM+. You can perform that registration in three ways:

- Manually, using a command line utility called *RegSvcs.exe*.
- Dynamically, by having the client program register your assembly automatically.
- Programmatically, by writing code that does the registration for you using a utility class provided by .NET.

Regardless of the technique you use, the registration process adds your serviced components to a COM+ application and configures them according to the default COM+ settings or according to their attributes (if present in the code). If the assembly contains incompatible attributes, the incompatibility is detected during registration and the registration is aborted. Future versions of the .NET compilers may detect incompatibilities during compilation time.

Signing Assembly and Assembly Location

To add an assembly to a COM+ application, the assembly must be signed (have a strong name) so the assembly resolver can map a client activation request to the corresponding assembly. Although in theory you need not install the assembly in the *global assembly cache* (GAC), in practice you should install it because the assembly DLL must be in a known location—either the system directory (for server applications that run in *DllHost*) or the hosting client process directory (if the client is not a COM+ server application). The other known location that the assembly resolver uses is the GAC. To maintain flexibility (to change from server to library application) and consistency, make sure you always install your serviced component assembly in the GAC.

10.3.1 Specifying Application Name

You can provide .NET with an assembly attribute, specifying the name of the COM+ application you would like your components to be part of, by using the `ApplicationName` assembly attribute:

```
[assembly: ApplicationName("MyApp")]
```

If you do not provide an application name, .NET uses the assembly name. The `ApplicationName` attribute (and the rest of the serviced components attributes) is defined in the `System.EnterpriseServices` namespace. You must add this namespace to your project references and reference that namespace in your assembly information file:

```
using System.EnterpriseServices;
```

10.3.2 Understanding Serviced Component Versions

Before exploring the three registration options, you need to understand the relationship between an assembly's version and COM+ components.

Every managed client of your assembly is built against the particular version of the assembly that contains your components, whether they are serviced or regular managed components. .NET zealously enforces version compatibility between the client's assembly and any other assembly it uses. The assembly's version is the product of its version number (major and minor numbers, such as 3.11) and the build and revision numbers. The version number is provided by the developer as an assembly attribute, and the build or revision numbers can be generated by the compiler—or the developer can provide them himself.

The semantics of the version and build or revision numbers tell .NET whether two particular assembly versions are compatible with each other, and which of the two assemblies is the latest. Assemblies are compatible if the version number is the same. The default is that different build and revision numbers do not indicate incompatibility, but a difference in either major or minor number indicates incompatibility. A client's manifest contains the version of each assembly it uses. At runtime, .NET loads for the client the latest compatible assemblies to use, and latest is defined using the build and revision numbers.

All this is fine while everything is under tight control of the .NET runtime. But how would .NET guarantee compatibility between the assembly's version and the configuration of the serviced components in the COM+ Catalog? The answer is via the COM+ component's ID.

The first time a serviced component is added to a COM+ application, the registration process generates a CLSID for it, based on a hash of the class definition and its assembly's version and strong name. Subsequent registration of the same assembly with an incompatible version is considered a new registration for that serviced component, and the component is given a new CLSID. This way, the serviced component's CLSID serves as its configuration settings version number. Existing managed clients do not interfere with one another because each gets to use the assembly version it was compiled with. Each managed client also uses a particular set of configuration parameters for the serviced components, captured with a different CLSID. When a managed client creates a serviced component, the .NET runtime creates for it a component from an assembly with a compatible version and applies the COM+ configuration of the matching CLSID.

10.3.3 Manual Registration

To register your component manually, use the *RegSvcs.exe* command-line utility. (In the future, Visual Studio.NET will probably allow you to invoke RegSvcs from the visual environment itself.) RegSvcs accepts as a parameter the name of the file containing your assembly's metadata. In a single DLL assembly, that file is simply the assembly file. If you do not specify as an assembly attribute the name of the COM+ application that should host your components, RegSvcs must be told that name explicitly as a command-line parameter, using the `/appname:` switch.

For example, if your single DLL assembly resides in *MyAssembly.dll* and you wish to add the serviced components in that assembly to the MyApp COM+ application, you would use RegSvcs in this manner:

```
RegSvcs.exe /appname:MyApp  MyAssembly.dll
```

The command-line application name is ignored if the assembly contains an application name.

In any case, you must create that COM+ application in the Component Services Explorer beforehand; otherwise, the previous command line will fail. You can instruct RegSvcs to create the application for you using the `/c` switch:

```
RegSvcs.exe  /c  MyApp  MyAssembly.dll
```

Or if the name is specified in the assembly:

```
RegSvcs.exe  /c  MyAssembly.dll
```

When using the `/c` switch, RegSvcs creates a COM+ application, names it accordingly, and adds the serviced components to it. If the Catalog already contains an application with that name, the registration fails.

You can also ask RegSvcs to try to find a COM+ application with that name and, if none is found, create one. This is done using the `/fc` switch:

```
RegSvcs.exe  /fc MyApp MyAssembly.dll
```

Or if the name is specified in the assembly:

```
RegSvcs.exe  /fc  MyAssembly.dll
```

If you don't specify a COM+ application name, either in the assembly or as a command-line parameter, RegSvcs uses the assembly name for the application name. If your assembly is called MyAssembly, RegSvcs adds the components to the MyAssembly

COM+ application. This behavior is the same for all the command-line switches.
By default, RegSvcs does not override the existing COM+ application (and its components) settings. If that assembly version is already registered with that COM+ application, then RegSvcs does nothing. If that version is not registered yet, it adds the new version and assigns new CLSIDs. Reconfiguring an existing version is done explicitly using the `/reconfig` switch:

```
RegSvcs.exe /reconfig /fc MyApp MyAssembly.dll
```

The `/reconfig` switch causes RegSvcs to reapply any application, component, interface, and method attributes found in the assembly to the existing version and use the COM+ default settings for the rest, thus reversing any changes you made using the Component Services Explorer.
When RegSvcs adds a serviced component to the COM+ Catalog, it must give it a class-ID (CLSID) and a prog-ID. RegSvcs creates a GUID for every component (based on the assembly's version and the class definition) and names it *<Namespace>*.*<Component name>*. For example, when you add the serviced component in Example 10-1 to the COM+ Catalog, RegSvcs names it *MyNamespace.MyComponent*. You can also specify the CLSID and the prog-ID of your serviced components using attributes.
In addition to adding the serviced components in the assembly to a COM+ application, RegSvcs creates a type library. This library contains interface and CoClass definitions to be used by nonmanaged clients (COM clients). The default type library filename is <*Assembly name*>*.tlb*—the name of the assembly with a *.tlb* extension.

10.3.4 Dynamic Registration

When a managed client creates a serviced component, the .NET runtime resolves which assembly version to use for that client. Next, the runtime verifies that the required version is registered with COM+. If it is not registered, the runtime installs it automatically. This process is called *dynamic registration.* As with RegSvcs, if the assembly contains an application name, then that name is used; if it does not, then the assembly's name is used for the COM+ application's name.
Note that only .NET clients can rely on having dynamic registration done when they instantiate a .NET serviced component. For COM clients, you must use the RegSvcs utility. Another limitation of dynamic registration is that serviced components in the assembly are configured according to the attributes in the assembly and the COM+ defaults. If you require configuring some services (such as events subscriptions) using the Component Services Explorer for your application to function properly, you must use RegSvcs to

register your components and provide the additional configuration using the Component Services Explorer. Only then can clients use your serviced components. As a result, dynamic registration is only useful for serviced components that contain all the service configurations they need in their code through the use of attributes. Finally, dynamic registration requires that the user invoking the call that triggers dynamic registration be a member of the Windows 2000 Administrator group. It has this requirement because dynamic registration makes changes to the COM+ Catalog; if the user invoking it is not a member of the Windows 2000 Administrator group, dynamic registration will fail.
In general, you should use RegSvcs and the Component Services Explorer rather than relying on dynamic registration. If you want to rely on dynamic registration of your serviced components, you should increment the version number of your assembly every time you make a change to one of the components' attributes, to ensure that you trigger dynamic registration.

10.3.5 Programmatic Registration

Both RegSvcs and dynamic registration use a .NET class called `RegistrationHelper` to perform the registration. `RegistrationHelper` implements the `IRegistrationHelper` interface, whose methods are used to register and unregister assemblies. For example, the `InstallAssembly( )` method registers the specified assembly in the specified COM+ application (or the application specified in the assembly). This method is defined as:

```
public void InstallAssembly(string assembly,
                            ref string application,
                            ref string tlb,
                            InstallationFlags
installFlags );
```

The installation flags correspond to the various RegSvcs switches. See the MSDN Library for additional information on `RegistrationHelper`. You can use `RegistrationHelper` yourself as part of your installation program; for more information, see Section 10.14 later in this chapter.

10.3.6 The ApplicationID Attribute

Every COM+ application has a GUID identifying it called the *application ID*. You can provide an assembly attribute specifying the application ID in addition to the application name:

```
[assembly: ApplicationID("8BE192FA-57D0-49a0-8608-
6829A314EEBE")]
```

Unlike the application name, the application ID is guaranteed to be unique, and you can use it alongside the application name. Once an application ID is specified, all searches for the application during registration are done using the application ID only, and the application name is only useful as a human-readable form of the application identity. Using application ID comes in handy when deploying the assembly in foreign markets—you can provide a command-line localized application name for every market while using the same application ID for your administration needs internally. The `ApplicationID` attribute is defined in the `System.EnterpriseServices` namespace.

10.3.7 The Guid Attribute

Instead of having the registration process generate a CLSID for your serviced component, you can specify one for it using the `Guid` attribute:

```
using System.Runtime.InteropServices;

[Guid("260C9CC7-3B15-4155-BF9A-12CB4174A36E")]
public class MyComponent :ServicedComponent,IMyInterface
{...}
```

The `Guid` attribute is defined in the `System.Runtime.InteropServices` namespace.
When you specify a class ID, subsequent registrations of the assembly don't generate a new CLSID for the component, regardless of the version of the assembly being registered. Registrations always reconfigure the same component in the COM+ Catalog. Specifying a class ID is useful during development, when you have multiple cycles of code-test-fix. Without it, every invocation by the test client triggers a dynamic registration—you very quickly clutter the COM+ application with dozens of components, when you actually only use the latest one.

10.3.8 The ProgId Attribute

Instead of having the registration process generate a name for your serviced component (namespace plus component name), you can specify one for it using the `ProgID` attribute:

```
using System.Runtime.InteropServices;

[ProgId("My Serviced Component")]
public class MyComponent :ServicedComponent,IMyInterface
{...}
```

The `ProgId` attribute is defined in the `System.Runtime.InteropServices` namespace.

10.4 Configuring Serviced Components

You can use various .NET attributes to configure your serviced components to take advantage of COM+ component services. The rest of this chapter demonstrates this service by service, according to the order in which the COM+ services are presented in this book.

10.5 Application Activation Type

To specify the COM+ application's activation type, you can use the `ApplicationActivation` assembly attributes. You can request that the application be a library or a server application:

```
[assembly:
ApplicationActivation(ActivationOption.Server)]
```

or:

```
[assembly:
ApplicationActivation(ActivationOption.Library)]
```

If you do not provide the `ApplicationActivation` attribute, then .NET uses a library activation type by default. Note that this use differs from the COM+ default of creating a new application as a server application.

> The next release of Windows 2000, Windows XP (see Appendix B), allows a COM+ application to be activated as a system service, so I expect that `ApplicationActivation` will be extended to include the value of `ActivationOption.Service`.

Before I describe other serviced components attributes, you need to understand what attributes are. Every .NET attribute is actually a class, and the attribute class has a constructor (maybe even a few overloaded constructors) and, usually, a few properties you can set. The syntax for declaring an attribute is different from that of any other class. In C#, you specify the attribute type between square brackets `[...]`. You specify constructor parameters and the values of the properties you wish to set between parentheses `(...)`.

In the case of the `ApplicationActivation` attribute, there are no properties and the constructor must accept an enum parameter of type `ActivationOption`, defined as:

```
enum ActivationOption{Server,Library}
```

There is no default constructor for the `ApplicationActivation` attribute.

The `ApplicationActivation` attribute is defined in the `System.EnterpriseServices` namespace. Your must add this namespace to your project references and reference that namespace in your assembly information file:

```
using System.EnterpriseServices;
```

The rest of this chapter assumes that you have added these references and will not mention them again.

> A client assembly that creates a serviced component or uses any of its base class `ServicedComponent` methods must add a reference to `System.EnterpriseServices` to its project. Other clients, which only use the interfaces provided by your serviced components, need not add the reference.

10.6 The Description Attribute

The `Description` attribute allows you to add text to the description field on the General Properties tab of an application, component, interface, or method. Example 10-2 shows how to apply the `Description` attribute at the assembly, class, interface, and method levels. After registration, the assembly-level description string becomes the content of the hosting COM+ application's description field; the class description string becomes the content of the COM+ component description field. The interface and method descriptions are mapped to the corresponding interface and method in the Component Services Explorer.

Example 10-2. Applying the Description attribute at the assembly, class, interface, and method levels

```
[assembly: Description("My Serviced Components
Application")]

[Description("IMyInterface description")]
public interface IMyInterface
{
   [Description("MyMethod description")]
   void MyMethod(  );
}

[Description("My Serviced Component description")]
public class MyComponent :ServicedComponent,IMyInterface
{
   public void MyMethod(  ){}
}
```

10.7 Accessing the COM+ Context

To access the COM+ context object's interfaces and properties, .NET provides you with the helper class `ContextUtil`. All context object interfaces (including the legacy MTS interfaces) are implemented as public static methods and public static properties of the `ContextUtil` class. Because the methods and properties are static, you do not have to instantiate a `ContextUtil` object—you should just call the methods. For example, if you want to trace the current COM+ context ID (its GUID) to the Output window, use the `ContextId` static property of `ContextUtil`:

```
using System.Diagnostics;//For the Trace class

Guid contextID = ContextUtil.ContextId;
String traceMessage = "Context ID is " +
contextID.ToString(  );
Trace.WriteLine(traceMessage);
```

`ContextUtil` has also properties used for JITA deactivation, transaction voting, obtaining the transactions and activity IDs, and obtaining the current transaction object. You will see examples for how to use these `ContextUtil` properties later in this chapter.

10.8 COM+ Context Attributes

You can decorate (apply attributes to) your class with two context-related attributes. The attribute `MustRunInClientContext` informs COM+ that the class must be activated in its creator's context:

```
[MustRunInClientContext(true)]
public class MyComponent :ServicedComponent
{...}
```

When you register the class above with COM+, the "Must be activated in caller's context" checkbox on the component's Activation tab is selected in the Component Services Explorer. If you do not use this attribute, the registration process uses the default COM+ setting when registering the component with COM+—not enforcing same-context activation. As a result, using `MustRunInClientContext` with a `false` parameter passed to the constructor is the same as using the COM+ default:

```
[MustRunInClientContext(false)]
```

Using attributes with the COM+ default values (such as constructing the `MustRunInClientContext` attribute with `false`) is useful when you combine it with the `/reconfig` switch of RegSvcs. For example, you can undo any unknown changes made to your component configuration using the Component Services Explorer and restore the component configuration to a known state.

The `MustRunInClientContext` attribute class has an overloaded default constructor. If you use `MustRunInClientContext` with no parameters, the default constructor uses `true` for the attribute value. As a result, the following two statements are equivalent:

```
[MustRunInClientContext]
[MustRunInClientContext(true)]
```

The second COM+ context-related attribute is the `EventTrackingEnabled` attribute. It informs COM+ that the component supports events and statistics collection during its execution:

```
[EventTrackingEnabled(true)]
public class MyComponent2:ServicedComponent
{...}
```

The statistics are displayed in the Component Services Explorer. When you register this class with COM+, the "Component supports events and statistics" checkbox on the component's Activation tab is checked in the Component Services Explorer. If you do not use this attribute, the registration process does not use the default COM+ setting of supporting events when registering the component with COM+. The .NET designers made this decision consciously to minimize creation of new COM+ contexts for new .NET components; a component that supports statistics is usually placed in it own context.

The `EventTrackingEnabled` attribute class also has an overloaded default constructor. If you construct it with no parameters, the default constructor uses `true` for the attribute value. As a result, the following two statements are equivalent:

```
[EventTrackingEnabled]
[EventTrackingEnabled(true)]
```

10.9 COM+ Object Pooling

The `ObjectPooling` attribute is used to configure every aspect of your component's object pooling. The `ObjectPooling` attribute enables or disables object pooling and sets the minimum or maximum pool size and object creation timeout. For example, to enable object pooling of your component's objects with a minimum pool size of 3, a maximum pool size of 10, and a creation timeout of 20 milliseconds, you would write:

```
[ObjectPooling(MinPoolSize = 3,MaxPoolSize =
10,CreationTimeout = 20)]
public class MyComponent :ServicedComponent
{...}
```

The `MinPoolSize`, `MaxPoolSize`, and `CreationTimeout` properties are public properties of the `ObjectPooling` attribute class. If you do not specify values for these properties (all or just a subset) when

your component is registered, the default COM+ values are used for these properties (a minimum pool size of 0, a maximum pool size of 1,048,576, and a creation timeout of 60 seconds).
The `ObjectPooling` attribute has a Boolean property called the `Enabled` property. If you do not specify a value for it (`true` or `false`), the attribute's constructor sets it to `true`. In fact, the attribute's constructor has a few overloaded versions—a default constructor that sets the `Enabled` property to `true` and a constructor that accepts a Boolean parameter. All constructors set the pool parameters to the default COM+ value. As a result, the following three statements are equivalent:

```
[ObjectPooling]
[ObjectPooling(true)]
[ObjectPooling(Enabled = true)]
```

> If your pooled component is hosted in a library application, then each hosting Application Domain will have its own pool. As a result, you may have multiple pools in a single physical process, if that process hosts multiple Application Domains.

Under COM, the pooled object returns to the pool when the client releases its reference to it. Managed objects do not have reference counting—.NET uses garbage collection instead. A managed pooled object returns to the pool only when it is garbage collected. The problem with this behavior is that a substantial delay between the time the object is no longer needed by its client and the time the object returns to the pool can occur. This delay may have serious adverse effects on your application scalability and throughput. An object is pooled because it was expensive to create. If the object spends a substantial portion of its time waiting for the garbage collector, your application benefits little from object pooling.
There are two ways to address this problem. The first solution uses COM+ JITA (discussed next). When you use JITA, the pooled object returns to the pool after every method call from the client. The second solution requires client participation.
ServicedComponent has a public static method called `DisposeObject( )`, defined as:

```
public static void DisposeObject(ServicedComponent sc);
```

When the client calls `DisposeObject( )`, passing in an instance of a pooled serviced component, the object returns to the pool immediately. `DisposeObject( )` has the effect of notifying COM+ that the object has been released. Besides returning the object to the pool, `DisposeObject( )` disposes of the context object hosting the pooled object and of the proxy the client used.
For example, if the component definition is:

```
public interface IMyInterface
{
   void MyMethod(  );
}

[ObjectPooling]
public class MyComponent : ServicedComponent,IMyInterface
{
   public void MyMethod(  ){}
}
```

When the client is done using the object, to expedite returning the object to the pool, the client should call `DisposeObject( )`:

```
IMyInterface obj;
Obj = (IMyInterface) new MyComponent(  );
obj.MyMethod(  );
ServicedComponent sc = obj as ServicedComponent;
If(sc != null)
      ServicedComponent.DisposeObject(sc);
```

However, calling `DisposeObject( )` directly is ugly. First, the client has to know that it is dealing with an object derived from `ServicedComponent`, which couples the client to the type used and renders many benefits of interface-based programming useless. Even worse, the client only has to call `DisposeObject( )` if this object is pooled, which couples the client to the serviced component's configuration. What if you use object pooling in only one customer site, but not in others? This situation is a serious breach of encapsulation—the core principle of object-oriented programming.
The solution is to have `ServicedComponent` implement a special interface (defined in the `System` namespace) called `IDisposable`, defined as:

```
public interface IDisposable
{
    void Dispose(  );
}
```

`ServicedComponent` implementation of `Dispose( )` returns the pooled object to the pool.
Having the `Dispose( )` method on a separate interface allows the client to query for the presence of `IDisposable` and always call it, regardless of the object's actual type:

```
IMyInterface obj;
obj = (IMyInterface) new MyComponent(  );
obj.MyMethod(  );

//Client wants to expedite whatever needs expediting:
IDisposable disposable = obj as IDisposable;
if(disposable != null)
   disposable.Dispose(  );
```

The `IDisposable` technique is useful not only with serviced components, but also in numerous other places in .NET. Whenever your component requires deterministic disposal of the resources and memory it holds, `IDisposable` provides a type-safe, component-oriented way of having the client dispose of the object without being too coupled to its type.

10.10 COM+ Just-in-Time Activation

.NET managed components can use COM+ JITA to efficiently handle rich clients (such as .NET Windows Forms clients), as discussed in Chapter 3.

To enable JITA support for your component, use the `JustInTimeActivation` attribute:

```
[JustInTimeActivation(true)]
public class MyComponent :ServicedComponent
{..}
```

When you register this component with COM+, the JITA checkbox in the Activation tab on the Component Services Explorer is selected. If you do not use the `JustInTimeActivation` attribute, JITA support is disabled when you register your component with COM+ (unlike the COM+ default of enabling JITA). The `JustInTimeActivation` class default constructor enables JITA support, so the following two statements are equivalent:

```
[JustInTimeActivation]
[JustInTimeActivation (true)]
```

Enabling JITA support is just one thing you need to do to use JITA. You still have to let COM+ know when to deactivate your object. You can deactivate the object by setting the done bit in the context object, using the `DeactivateOnReturn` property of the `ContextUtil` class. As discussed at length in Chapter 3, a JITA object should retrieve its state at the beginning of every method call and save it at the end. Example 10-3 shows a serviced component using JITA.

Example 10-3. A serviced component using JITA

```
public interface IMyInterface
{
   void MyMethod(long objectIdentifier);
}

[JustInTimeActivation(true)]
public class MyComponent :ServicedComponent,IMyInterface
{
   public void MyMethod(long objectIdentifier)
   {
      GetState(objectIdentifier);
      DoWork(  );
      SaveState(objectIdentifier);
```

```
        //inform COM+ to deactivate the object upon
method return
      ContextUtil.DeactivateOnReturn = true;
   }
   //other methods
   protected void GetState(long objectIdentifier){...}
   protected void DoWork(  ){...}
   protected void SaveState(long objectIdentifier){...}
}
```

You can also use the Component Services Explorer to configure the method to use auto-deactivation. In that case, the object is deactivated automatically upon method return, unless you set the value of the `DeactivateOnReturn` property to `false`.

10.10.1 Using IObjectControl

If your serviced component uses object pooling or JITA (or both), it may also need to know when it is placed in a COM+ context to do context-specific initialization and cleanup. Like a COM+ configured component, the serviced component can use `IObjectControl` for that purpose. The .NET base class `ServicedComponent` already implements `IObjectControl`, and its implementation is virtual—so you can override the implementation in your serviced component, as shown in Example 10-4.

Example 10-4. A serviced component overriding the ServicedComponent implementation of IObjectControl

```
public class MyComponent :ServicedComponent
{
   public override void Activate(  )
   {
      //Do context specific initialization here
   }
   public override void Deactivate(  )
   {
      //Do context specific cleanup here
   }
   public override bool CanBePooled(  )
   {
          return true;
   }
   //other methods
}
```

If you encounter an error during `Activate( )` and throw an exception, then the object's activation fails and the client is given an opportunity to catch the exception.

10.10.2 IObjectControl, JITA, and Deterministic Finalization

To maintain JITA semantics, when the object deactivates itself, .NET calls `DisposeObject( )` on it explicitly, thus destroying it. Your object can do specific cleanup in the `Finalize( )` method (the destructor in C#), and `Finalize( )` will be called as soon as the object deactivates itself, without waiting for garbage collection. If the object is a pooled object (as well as a JITA object), then it is returned to the pool after deactivation, without waiting for the garbage collection.

You can also override the `ServicedComponent` implementation of `IObjectControl.Deactivate( )` and perform your cleanup there. In any case, you end up with a deterministic way to dispose of critical resources without explicit client participations. This situation makes sharing your object among clients much easier because now the clients do not have to coordinate who is responsible for calling `Dispose( )`.

COM+ JITA gives managed components deterministic finalization, a service that nothing else in .NET can provide out of the box.

10.11 COM+ Constructor String

Any COM+ configured component that implements the `IObjectConstruct` interface has access during construction to a construction string (discussed in Chapter 3), configured in the Component Services Explorer. Serviced components are no different. The base class, `ServicedComponent`, already implements the `IObjectConstruct` interface as a virtual method (it has only one method). Your derived serviced component can override the `Construct( )` method, as shown in this code sample:

```
public class MyComponent :ServicedComponent
{
   public override void Construct(string constructString)
   {
      //use the string. For example:
      MessageBox.Show(constructString);
   }
}
```

If the checkbox "Enable object construction" on the component Activation tab is selected, then the `Construct( )` method is called after the component's constructor, providing it with the configured construction string.

You can also enable construction string support and provide a default construction string using the `ConstructionEnabled` attribute:

```
[ConstructionEnabled(Enabled = true,Default = "My
String")]
public class MyComponent :ServicedComponent
{
   public override void Construct(string constructString)
   {...}
}
```

The `ConstructionEnabled` attribute has two public properties. `Enabled` enables construction string support for your serviced component in the Component Services Explorer (once the component is registered) and `Default` provides an initial string value. When your component is registered with COM+, the registration process assigns the default string to the constructor string field on the component Activation tab. The default string has no further use after registration. New instances of your component receive as a constructor string the current value of the constructor string field. For example, if the default string is String A, when the serviced component is registered, the value of the constructor string field is set to String A. If you set it to a different value, such as String B, new instances of the component get String B as their construction string. They receive the current value, not the default value.

The `ConstructionEnabled` attribute has two overloaded constructors. One constructor accepts a Boolean value for the `Enabled` property; the default constructor sets the value of the `Enabled` property to `true`. You can also set the value of the `Enabled` property explicitly. As a result, the following three statements are equivalent:

```
[ConstructionEnabled]
[ConstructionEnabled(true)]
[ConstructionEnabled(Enabled = true)]
```

10.12 COM+ Transactions

You can configure your serviced component to use the five available COM+ transaction support options by using the `Transaction` attribute. The `Transaction` attribute's constructor accepts an enum parameter of type `TransactionOption`, defined as:

```
public enum TransactionOption
{
   Disabled,
   NotSupported,
   Supported,
   Required,
   RequiresNew
}
```

For example, to configure your serviced component to require a transaction, use the `TransactionOption.Required` value:

```
[Transaction(TransactionOption.Required)]
public class MyComponent :ServicedComponent
{...}
```

The five enum values of `TransactionOption` map to the five COM+ transaction support options discussed in Chapter 4.

When you use the `Transaction` attribute to mark your serviced component to use transactions, you implicitly set it to use JITA and require activity-based synchronization as well.

The `Transaction` attribute has an overloaded default constructor, which sets the transaction support to `TransactionOption.Required`. As a result, the following two statements are equivalent:

```
[Transaction]
[Transaction(TransactionOption.Required)]
```

10.12.1 Voting on the Transaction

Not surprisingly, you use the `ContextUtil` class to vote on the transaction's outcome. `ContextUtil` has a static property of the enum type `TransactionVote` called `MyTransactionVote`. `TransactionVote` is defined as:

```
public enum TransactionVote {Abort,Commit}
```

Example 10-5 shows a transactional serviced component voting on its transaction outcome using `ContextUtil`. Note that the component still has to do all the right things that a well-designed transactional component has to do (see Chapter 4); it needs to retrieve its state from a resource manager at the beginning of the call and save it at the end. It must also deactivate itself at the end of the method to purge its state and make the vote take effect.

Example 10-5. A transactional serviced component voting on its transaction outcome using the ContextUtil MyTransactionVote property

```
public interface IMyInterface
{
   void MyMethod(long objectIdentifier);
}

[Transaction]
public class MyComponent :ServicedComponent,IMyInterface
{
   public void MyMethod(long objectIdentifier)
   {
      try
      {
         GetState(objectIdentifier);
```

```
          DoWork(  );
          SaveState(objectIdentifier);
          ContextUtil.MyTransactionVote =
TransactionVote.Commit;
       }
       catch
       {
          ContextUtil.MyTransactionVote =
TransactionVote.Abort;
       }
       //Let COM+ deactivate the object once the method
returns
       finally
       {
          ContextUtil.DeactivateOnReturn = true;
       }
   }
   //helper methods
   protected void GetState(long objectIdentifier){...}
   protected void DoWork(  ){...}
   protected void SaveState(long objectIdentifier){...}
}
```

Compare Example 10-5 to Example 4-3. A COM+ configured component uses the returned `HRESULT` from the `DoWork( )` helper method to decide on the transaction's outcome. A serviced component, like any other managed component, does not use `HRESULT` return codes for error handling; it uses exceptions instead. In Example 10-5 the component catches any exception that was thrown in the `try` block by the `DoWork( )` method and votes to abort in the `catch` block.

Alternatively, if you do not want to write exception-handling code, you can use the programming model shown in Example 10-6. Set the context object's consistency bit to `false` (vote to abort) as the first thing the method does. Then set it back to `true` as the last thing the method does (vote to commit). Any exception thrown in between causes the method exception to end without voting to commit.

Example 10-6. Voting on the transaction without exception handling

```
public interface IMyInterface
{
   void MyMethod(long objectIdentifier);
}

[Transaction]
public class MyComponent :ServicedComponent,IMyInterface
{
   public void MyMethod(long objectIdentifier)
   {
       //Let COM+ deactivate the object once the method
returns and abort the
       //transaction. You can use ContextUtil.SetAbort(
) as well
       ContextUtil.DeactivateOnReturn = true;
       ContextUtil.MyTransactionVote =
TransactionVote.Abort;

       GetState(objectIdentifier);
       DoWork(  );
       SaveState(objectIdentifier);

       ContextUtil.MyTransactionVote =
TransactionVote.Commit;
   }
   //helper methods
   protected void GetState(long objectIdentifier){...}
   protected void DoWork(  ){...}
   protected void SaveState(long objectIdentifier){...}
}
```

Example 10-6 has another advantage over Example 10-5: having the exception propagated up the call chain once the transaction is aborted. By propagating it, callers up the chain know that they can also abort their work and avoid wasting more time on a doomed transaction.

10.12.2 The AutoComplete Attribute

Your serviced components can take advantage of COM+ method auto-deactivation using the `AutoComplete` method attribute. During the registration process, the method is configured to use COM+ auto-deactivation when `AutoComplete` is used on a method, and the checkbox "Automatically deactivate this object when the method returns" on the method's General tab is selected. Serviced components that use the `AutoComplete` attribute do not need to vote explicitly on their transaction outcome. Example 10-7 shows a transactional serviced component using the `AutoComplete` method attribute.

Example 10-7. Using the AutoComplete method attribute

```
public interface IMyInterface
{
   void MyMethod(long objectIdentifier);
}

[Transaction]
public class MyComponent : ServicedComponent,IMyInterface
```

```
{
   [AutoComplete(true)]
   public void MyMethod(long objectIdentifier)
   {
       GetState(objectIdentifier);
       DoWork(  );
       SaveState(objectIdentifier);
   }
   //helper methods
   protected void GetState(long objectIdentifier){...}
   protected void DoWork(  ){...}
   protected void SaveState(long objectIdentifier){...}
}
```

When you configure the method to use auto-deactivation, the object's interceptor sets the done and consistency bits of the context object to `true` if the method did not throw an exception and the consistency bit to `false` if it did. As a result, the transaction is committed if no exception is thrown and aborted otherwise.

Nontransactional JITA objects can also use the `AutoComplete` attribute to deactivate themselves automatically on method return.

The `AutoComplete` attribute has an overloaded default constructor that uses `true` for the attribute construction. Consequently, the following two statements are equivalent:

```
[AutoComplete]
[AutoComplete(true)]
```

The `AutoComplete` attribute can be applied on a method as part of an interface definition:

```
public interface IMyInterface
{
   //Avoid this:
   [AutoComplete]
   void MyMethod(long objectIdentifier);
}
```

However, you should avoid using the attribute this way. An interface and its methods declarations serve as a contract between a client and an object; using auto completion of methods is purely an implementation decision. For example, one implementation of the interface on one component may chose to use autocomplete and another implementation on another component may choose not to.

10.12.3 The TransactionContext Object

A nontransactional managed client creating a few transactional objects faces a problem discussed in Chapter 4 (see Section 4.9). Essentially, if the client wants to scope all its interactions with the objects it creates under one transaction, it must use a middleman to create the objects for it. Otherwise, each object created will be in its own separate transaction. COM+ provides a ready-made middleman called TransactionContext. Managed clients can use TransactionContext as well. To use the TransactionContext object, add to the project references the COM+ services type library. The TransactionContext class is in the COMSVCSLib namespace.

The TransactionContext class is especially useful in situations in which the class is a managed .NET component that derives from a class other than ServicedComponent. Remember that a .NET component can only derive from one concrete class and since the class already derives from a concrete class other than ServicedComponent, it cannot use the Transaction attribute. Nevertheless, the TransactionContext class gives this client an ability to initiate and manage a transaction.

Example 10-8 demonstrates usage of the TransactionContext class, using the same use-case as Example 4-6.

Example 10-8. A nontransactional managed client using the TransactionContext helper class to create other transactional objects

```
using COMSVCSLib;

IMyInterface obj1,obj2,obj3;
ITransactionContext transContext;

transContext = (ITransactionContext) new
TransactionContext(  );

obj1 =
(IMyInterface)transContext.CreateInstance("MyNamespace.My
Component");
obj2 =
(IMyInterface)transContext.CreateInstance("MyNamespace.My
Component");
obj3 =
(IMyInterface)transContext.CreateInstance("MyNamespace.My
Component");

try
{
   obj1.MyMethod(  );
   obj2.MyMethod(  );
   obj3.MyMethod(  );
   transContext.Commit(  );
}
catch//Any error - abort the transaction
{
   transContext.Abort(  );
}
```

Note that the client in Example 10-8 decides whether to abort or commit the transaction depending on whether an exception is thrown by the internal objects.

10.12.4 COM+ Transactions and Nonserviced Components

Though this chapter focuses on serviced components, it is worth noting that COM+ transactions are used by other parts of the .NET framework besides serviced components—in particular, ASP.NET and Web Services.

10.12.4.1 Web services and transactions

Web services are the most exciting piece of technology in the entire .NET framework. *Web services* allow a middle-tier component in one web site to invoke methods on another middle-tier component at another web site, with the same ease as if that component were in its own assembly. The underlying technology facilitating web services serializes the calls into text format and transports the call from the client to the web service provider using HTTP. Because web service calls are text based, they can be made across firewalls. Web services typically use a protocol called Simple Object Access Protocol (SOAP) to represent the call, although other text-based protocols such as HTTP-POST and HTTP-GET can also be used. .NET successfully hides the required details from the client and the server developer; a web service developer only needs to use the `WebMethod` attribute on the public methods exposed as web services. Example 10-9 shows the `MyWebService` web service that provides the `MyMessage` web service—it returns the string "Hello" to the caller.

Example 10-9. A trivial web service that returns the string "Hello"

```
using System.Web.Services;

public class MyWebService : WebService
{
    public MyWebService(  ){}
    [WebMethod]
    public string MyMessage(  )
    {
       return "Hello";
    }
}
```

The web service class can optionally derive from the `WebService` base class, defined in the `System.Web.Services` namespace (see Example 10-9). The `WebService` base class provides you with easy access to common ASP.NET objects, such as those representing

application and session states. Your web service probably accesses resource managers and transactional components. The problem with adding transaction support to a web service that derived from `WebService` is that it is not derived from `ServicedComponent`, and .NET does not allow multiple inheritance of implementation.
To overcome this hurdle, the `WebMethod` attribute has a public property called `TransactionOption`, of the enum type `Enterprise.Services.TransactionOption` discussed previously. The default constructor of the `WebMethod` attribute sets this property to `TransactionOption.Disabled`, so the following two statements are equivalent:

```
[WebMethod]
[WebMethod(TransactionOption =
TransactionOption.Disabled)]
```

If your web service requires a transaction, it can only be the root of a transaction, due to the stateless nature of the HTTP protocol. Even if you configure your web method to only require a transaction and it is called from within the context of an existing transaction, a new transaction is created for it. Similarly, the value of `TransactionOption.Supported` does not cause a web service to join an existing transaction (if called from within one). Consequently, the following statements are equivalent—all four amount to no transaction support for the web service:

```
[WebMethod]
[WebMethod(TransactionOption =
TransactionOption.Disabled)]
[WebMethod(TransactionOption =
TransactionOption.NotSupported)]
[WebMethod(TransactionOption =
TransactionOption.Supported)]
```

Moreover, the following statements are also equivalent—creating a new transaction for the web service:

```
[WebMethod(TransactionOption =
TransactionOption.Required)]
[WebMethod(TransactionOption =
TransactionOption.RequiresNew)]
```

The various values of `TransactionOption` are confusing. To avoid making them the source of errors and misunderstandings, use `TransactionOption.RequiresNew` when you want transaction support for your web method; use `TransactionOption.Disabled` when you want to explicitly demonstrate to a reader of your code that the web service does not take part in a transaction. The question is, why did Microsoft provide four overlapping transaction modes for web services? I believe that it is not the result of carelessness, but rather a conscious design decision. Microsoft is probably laying down the foundation in .NET for a point in the future when it will be possible to propagate transactions across web sites.

Finally, you do not need to explicitly vote on a transaction from within a web service. If an exception occurs within a web service method, the transaction is automatically aborted. Conversely, if no exceptions occur, the transaction is committed automatically (as if you used the `AutoComplete` attribute). Of course, the web service can still use `ContextUtil` to vote explicitly to abort instead of throwing an exception, or when no exception occurred and you still want to abort.

10.12.4.2 ASP.NET and transactions

An ASP.NET web form may access resource managers (such as databases) directly, and it should do so under the protection of a transaction. The page may also want to create a few transactional components and compose their work into a single transaction. The problem again is that a web form derives from the `System.Web.UI.Page` base class, not from `ServicedComponent`, and therefore cannot use the `[Transaction]` attribute.

To provide transaction support for a web form, the Page base class has a write-only property called `TransactionMode` of type `TransactionOption`. You can assign a value of type `TransactionOption` to `TransactionMode`, to configure transaction support for your web form. You can assign `TransactionMode` programmatically in your form contractor, or declaratively by setting that property in the visual designer. The designer uses the *Transaction page directive* to insert a directive in the *aspx* form file. For example, if you set the property using the designer to `RequiresNew`, the designer added this line to the beginning of the *aspx* file:

```
<@% Page Transaction="RequiresNew" %>
```

Be aware that programmatic setting will override any designer setting. The default is no transaction support (disabled).

The form can even vote on the outcome of the transaction (based on its interaction with the components it created) by using the `ContextUtil` methods. Finally, the form can subscribe to events notifying it when a transaction is initiated and when a transaction is aborted.

10.13 COM+ Synchronization

Multithreaded managed components can use .NET-provided synchronization locks. These are classic locks, such as mutexes and events. However, these solutions all suffer from the deficiencies described at the beginning of Chapter 5. .NET serviced components should use COM+ activity-based synchronization by adding the `Synchronization` attribute to the class definition. The

`Synchronization` attribute's constructor accepts an enum parameter of type `SynchronizationOption`, defined as:

```
public enum SynchronizationOption
{
    Disabled,
    NotSupported,
    Supported,
    Required,
    RequiresNew
}
```

For example, use the `SynchronizationOption.Required` value to configure your serviced component to require activity-based synchronization:

```
[Synchronization(SynchronizationOption.Required)]
public class MyComponent :ServicedComponent
{...}
```

The five enum values of `SynchronizationOption` map to the five COM+ synchronization support options discussed in Chapter 5.

The `Synchronization` attribute has an overloaded default constructor, which sets synchronization support to `SynchronizationOption.Required`. As a result, the following two statements are equivalent:

```
[Synchronization]
[Synchronization(SynchronizationOption.Required)]
```

> The `System.Runtime.Remoting.Context` namespace contains a context attribute called `Synchronization` that can be applied to context-bound .NET classes. This attribute accepts synchronization flags similar to `SynchronizationOption`, and initially looks like another version of the `Synchronization` class attribute. However, the `Synchronization` attribute in the `Context` namespace provides synchronization based on physical threads, unlike the `Synchronization` attribute in the `EnterpriseServices` namespace, which uses causalities. As explained in Chapter 5, causality and activities are a more elegant and fine-tuned synchronization strategy.

10.14 Programming the COM+ Catalog

You can access the COM+ Catalog from within any .NET managed component (not only serviced components). To write installation or configuration code (or manage COM+ events), you need to add to your project a reference to the COM+ Admin type library. After you add the reference, the Catalog interfaces and objects are part of the `COMAdmin` namespace. Example 10-10 shows how to create a catalog object and use it to iterate over the application collection, tracing to the Output window the names of all COM+ applications on your computer.

Example 10-10. Accessing the COM+ Catalog and tracing the COM+ application names

```
using COMAdmin;

ICOMAdminCatalog catalog;
ICatalogCollection applicationCollection;
ICatalogObject application;

int applicationCount;
int i;//Application index

catalog = (ICOMAdminCatalog)new COMAdminCatalog(  );
applicationCollection =
(ICatalogCollection)catalog.GetCollection("Applications")
;

//Read the information from the catalog
applicationCollection.Populate(  );
applicationCount = applicationCollection.Count;

for(i = 0;i< applicationCount;i++)
{
   //Get the current application
   application=
(ICatalogObject)applicationCollection.get_Item(i);
   int index = i+1;
   String traceMessage = index.ToString()+".
"+application.Name.ToString(  );

   Trace.WriteLine(traceMessage);
}
```

The `System.EnterpriseServices.Admin` namespace contains the COM+ Catalog object and interface definitions. However, in the Visual Studio.NET Beta 2, the interfaces are defined as private to that assembly. As a result, you cannot access them. The obvious workaround is to import the COM+ Admin type library yourself, as demonstrated in Example 10-10. In the future, you will probably be able to use `System.EnterpriseServices.Admin` namespace directly. The resulting code, when programming directly using the `System.EnterpriseServices.Admin` namespace, is almost identical to Example 10-10.

10.15 COM+ Security

.NET has an elaborate component-oriented security model. .NET security model manages what the component is allowed to do and what permissions are given to the component and all its clients up the call chain. You can (and should) still manage the security attributes of your hosting COM+ application to authenticate incoming calls, authorize callers, and control impersonation level. .NET also has what .NET calls role-based security, but that service is limited compared with COM+ role-based security. A *role* in .NET is actually a Windows NT user group. As a result, .NET role-based security is only as granular as the user groups in the hosting domain. Usually, you do not have control over your end customer's IT department. If you deploy your application in an environment where the user groups are coarse, or where they do not map well to actual roles users play in your application, then .NET role-based security is of little use to you. COM+ roles are unrelated to the user groups, allowing you to assign roles directly from the application business domain.

10.15.1 Configuring Application-Level Security Settings

The assembly attribute `ApplicationAccessControl` is used to configure all the settings on the hosting COM+ application's Security tab.

You can use `ApplicationAccessControl` to turn application-level authentication on or off:

```
[assembly: ApplicationAccessControl(true)]
```

The `ApplicationAccessControl` attribute has a default constructor, which sets authorization to `true` if you do not provide a construction value. Consequently, the following two statements are equivalent:

```
[assembly: ApplicationAccessControl]
[assembly: ApplicationAccessControl(true)]
```

If you do not use the `ApplicationAccessControl` attribute at all, then when you register your assembly, the COM+ default takes effect and application-level authorization is turned off.

The `ApplicationAccessControl` attribute has three public properties you can use to set the access checks, authentication, and impersonation level. The `AccessChecksLevel` property accepts an enum parameter of type `AccessChecksLevelOption`, defined as:

```
public enum AccessChecksLevelOption
{
    Application,
    ApplicationComponent
}
```

`AccessChecksLevel` is used to set the application-level access checks to the process only (`AccessChecksLevelOption.Application`) or process and component level (`AccessChecksLevelOption.ApplicationComponent`). If you do not specify an access level, then the `ApplicationAccessControl` attribute's constructors set the access level to `AccessChecksLevelOption.ApplicationComponent`, the same as the COM+ default.

The `Authentication` property accepts an enum parameter of type `AuthenticationOption`, defined as:

```
public enum AuthenticationOption
{
   None,
   Connect,
   Call,
   Packet,
   Integrity,
   Privacy,
   Default
}
```

The values of `AuthenticationOption` map to the six authentication options discussed in Chapter 7. If you do not specify an authentication level or if you use the `Default` value, the `ApplicationAccessControl` attribute's constructors set the authentication level to `AuthenticationOption.Packet`, the same as the COM+ default.

The `Impersonation` property accepts an enum parameter of type `ImpersonationLevelOption`, defined as:

```
public enum ImpersonationLevelOption
{
   Anonymous,
   Identify,
   Impersonate,
   Delegate,
   Default
}
```

The values of `ImpersonationLevelOption` map to the four impersonation options discussed in Chapter 7. If you do not specify an impersonation level or if you use the `Default` value, then the `ApplicationAccessControl` attribute's constructors set the

impersonation level to ImpersonationLevelOption.Impersonate, the same as the COM+ default.
Example 10-11 demonstrates using the ApplicationAccessControl attribute with a server application. The example enables application-level authentication and sets the security level to perform access checks at the process and component level. It sets authentication to authenticate incoming calls at the packet level and sets the impersonation level to Identify.

Example 10-11. Configuring a server application security

```
[assembly:
ApplicationActivation(ActivationOption.Server)]

[assembly: ApplicationAccessControl(
           true,//Authentication is on

AccessChecksLevel=AccessChecksLevelOption.ApplicationComp
onent,
           Authentication=AuthenticationOption.Packet,

ImpersonationLevel=ImpersonationLevelOption.Identify)]
```

A library COM+ application has no use for impersonation level, and it can only choose whether it wants to take part in its hosting process authentication level (that is, it cannot dictate the authentication level). To turn authentication off for a library application, set the authentication property to AuthenticationOption.None. To turn it on, use any other value, such as AuthenticationOption.Packet. Example 10-12 demonstrates how to use the ApplicationAccessControl to configure the security setting of a library application.

Example 10-12. Configuring a library application security

```
[assembly:
ApplicationActivation(ActivationOption.Library)]

[assembly: ApplicationAccessControl(
           true,//Authentication

AccessChecksLevel=AccessChecksLevelOption.ApplicationComp
onent,
           //use AuthenticationOption.None to turn off
authentication,
           //and any other value to turn it on
           Authentication=AuthenticationOption.Packet)]
```

10.15.2 Component-Level Access Checks

The component attribute `ComponentAccessControl` is used to enable or disable access checks at the component level. Recall from Chapter 7 that this is your component's role-based security master switch. The `ComponentAccessControl` attribute's constructor accepts a Boolean parameter, used to turn access control on or off. For example, you can configure your serviced component to require component-level access checks:

```
[ComponentAccessControl(true)]
public class MyComponent :ServicedComponent
{...}
```

The `ComponentAccessControl` attribute has an overloaded default constructor that uses `true`for the attribute construction. Consequently, the following two statements are equivalent:

```
[ComponentAccessControl]
[ComponentAccessControl(true)]
```

10.15.3 Adding Roles to an Application

You can use the Component Services Explorer to add roles to the COM+ application hosting your serviced components. You can also use the `SecurityRole` attribute to add the roles at the assembly level. When you register the assembly with COM+, the roles in the assembly are added to the roles defined for the hosting COM+ application. For example, to add the Manager and Teller roles to a bank application, simply add the two roles as assembly attributes:

```
[assembly: SecurityRole("Manager")]
[assembly: SecurityRole("Teller")]
```

The `SecurityRole` attribute has two public properties you can set. The first is `Description`*.* Any text assigned to the `Description` property will show up in the Component Services Explorer in the Description field on the role's General tab:

```
[assembly: SecurityRole("Manager",Description = "Can
access all components")]
[assembly: SecurityRole("Teller",Description = "Can
access IAccountsManager only")]
```

The second property is the `SetEveryoneAccess` Boolean property. If you set `SetEveryoneAccess` to `true`, then when the component is registered, the registration process adds the user Everyone as a user for that role, thus allowing everyone access to whatever the role is assigned to. If you set it to `false`*,* then no user is added during registration and you have to explicitly add users during deployment using the Component Services Explorer. The `SecurityRole` attribute sets the value of `SetEveryoneAccess` by default to `true`. As a result, the following statements are equivalent:

```
[assembly: SecurityRole("Manager")]
[assembly: SecurityRole("Manager",true)]
```

```
[assembly: SecurityRole("Manager",SetEveryoneAccess =
true)]
```
Automatically granting everyone access is a nice debugging feature; it eliminates security problems, letting you focus on analyzing your domain-related bug. However, you must suppress granting everyone access in a release build, by setting the `SetEveryoneAccess` property to `false`:
```
#if DEBUG
[assembly: SecurityRole("Manager")]
#else
[assembly: SecurityRole("Manager",SetEveryoneAccess =
false)]
#endif
```

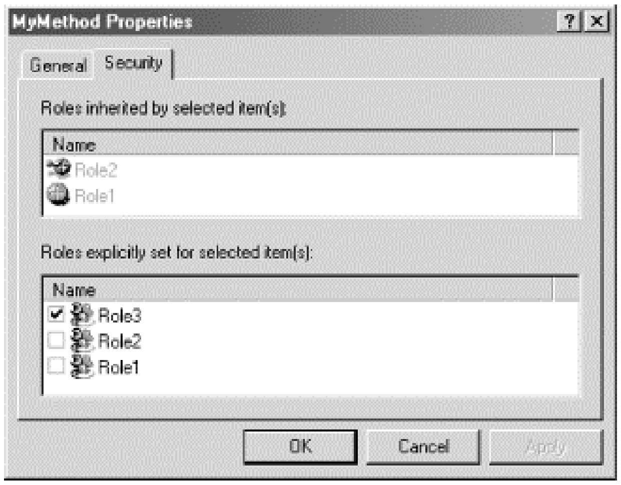
10.15.4 Assigning Roles to Component, Interface, and Method

The `SecurityRole` attribute is also used to grant access for a role to a component, interface, or method. Example 10-13 shows how to grant access to Role1 at the component level, to Role2 at the interface level, and to Role3 at the method level.

Example 10-13. Assigning roles at the component, interface, and method levels

```
[assembly: SecurityRole("Role1")]
[assembly: SecurityRole("Role2")]
[assembly: SecurityRole("Role3")]

[SecurityRole("Role2")]
public
interface IMyInterface
{
   [SecurityRole("Role3")]
   void MyMethod(  );
}

[SecurityRole("Role1")]
public class MyComponent :ServicedComponent,IMyInterface
{...}
```
Figure 10-2 shows the resulting role assignment in the Component Services Explorer at the method level. Note that Role1 and Role2 are inherited from the component and interface levels.

Figure 10-2. The resulting role assignment of Example 10-13 in the Component Services Explorer, as seen at the method level



Roles inherited by selected item(s):

Name
Role2
Role1

Roles explicitly set for selected item(s):

Name
✓ Role3
Role2
Role1

If you only assign a role (at the component, interface, or method level) but do not define it at the assembly level, then that role is added to the application automatically during registration. However, you should define roles at the assembly level to provide one centralized place for roles description and configuration.

10.15.5 Verifying Caller's Role Membership

Sometimes it is useful to verify programmatically the caller's role membership before granting it access. Your serviced components can do that just as easily as configured COM components. .NET provides you the helper class `SecurityCallContext` that gives you access to the security parameters of the current call. `SecurityCallContext` encapsulates the COM+ call-object's implementation of `ISecurityCallContext`, discussed in Chapter 7. The class `SecurityCallContext` has a public static property called `CurrentCall`. `CurrentCall` is a read-only property of type `SecurityCallContext` (it returns an instance of the same type). You use the `SecurityCallContext` object returned from `CurrentCall` to access the current call. Example 10-14 demonstrates the use of the security call context to verify a caller's role membership, using the same use-case as Example 7-1.

Example 10-14. Verifying the caller's role membership using the SecurityCallContext class

```
public class Bank :ServicedComponent,IAccountsManager
{
   void TransferMoney(int sum,ulong accountSrc,ulong
accountDest)
   {
      bool callerInRole = false;
      callerInRole =
SecurityCallContext.CurrentCall.IsCallerInRole("Customer"
);
```

```
         if(callerInRole)//The caller is a customer
      {
         if(sum > 5000)
           throw(new UnauthorizedAccessException(@"Caller
does not have sufficient
                                              credentials
to transfer this sum"));
      }
      DoTransfer(sum,accountSrc,accountDest);//Helper
method
   }
   //Other methods
}
```

You should use the Boolean property `IsSecurityEnabled` of `SecurityCallContext` to verify that security is enabled before accessing the `IsCallerInRole( )` method:

```
bool securityEnabled =
SecurityCallContext.CurrentCall.IsSecurityEnabled;
if(securityEnabled)
{
   //the rest of the verification process
}
```

10.16 COM+ Queued Components

.NET has a built-in mechanism for invoking a method call on an object: using a delegate asynchronously. The client creates a delegate class that wraps the method it wants to invoke synchronously, and the compiler provides definition and implementation for a `BeginInvoke( )` method, which asynchronously calls the required method on the object. The compiler also generates the `EndInvoke( )` method to allow the client to poll for the method completion. Additionally, .NET provides a helper class called `AsyncCallback` to manage asynchronous callbacks from the object once the call is done.
Compared with COM+ queued components, the .NET approach leaves much to be desired. First, .NET does not support disconnected work. Both the client and the server have to be running at the same time, and their machines must be connected to each other on the network. Second, the client's code in the asynchronous case is very different from the usual synchronous invocation of the same method on the object's interface. Third, there is no built-in support for transactional forwarding of calls to the server, nor is there an auto-retry mechanism. In short, you should use COM+ queued components if you want to invoke asynchronous method calls in .NET.

The ApplicationQueuing assembly attribute is used to configure queuing support for the hosting COM+ application. The ApplicationQueuing attribute has two public properties that you can set. The Boolean Enabledproperty corresponds to the Queued checkbox on the application's queuing tab. When set to true, it instructs COM+ to create a public message queue, named as the application, for the use of any queued components in the assembly. The second public property of ApplicationQueuing is the Boolean QueueListenerEnabled property. It corresponds to the Listen checkbox on the application's queuing tab. When set to true, it instructs COM+ to activate a listener for the application when the application is launched. For example, here is how you enable queued component support for your application and enable a listener:

```
//Must be a server application to use queued components
[assembly:
ApplicationActivation(ActivationOption.Server)]
[assembly: ApplicationQueuing(Enabled =
true,QueueListenerEnabled = true)]
```

The ApplicationQueuing attribute has an overloaded default constructor that sets the Enabled attribute to true and the QueueListenerEnabled attribute to false. As a result, the following two statements are equivalent:

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

10.16.1 Configuring Queued Interfaces

In addition to enabling queued component support at the application level, you must mark your interfaces as capable of receiving queued calls. You do that by using the InterfaceQueuing attribute. InterfaceQueuing has one public Boolean property called Enabled that corresponds to the Queued checkbox on the interface's Queuing tab.

```
[InterfaceQueuing(Enabled = true)]
public interface IMyInterface
{
   void MyMethod(  );
}
```

The InterfaceQueuing attribute has an overloaded default constructor that sets the Enabled property to true and a constructor that accepts a Boolean parameter. As a result, the following three statements are equivalent:

```
[InterfaceQueuing]
[InterfaceQueuing(true)]
[InterfaceQueuing(Enabled = true)]
```

Note that your interface must adhere to the queued components design guidelines discussed in Chapter 8, such as no `out` or `ref` parameters. If you configure your interface as a queued interface using the `InterfaceQueuing` attribute and the interface is incompatible with queuing requirements, the registration process fails.

10.16.2 A Queued Component's Managed Client

The client of a queued component cannot create the queued component directly. It must create a recorder for its calls using the `queue` moniker. A C++ or a Visual Basic 6.0 program uses the `CoGetObject( )` or `GetObject( )` calls. A .NET managed client can use the static method `BindToMoniker( )` of the `Marshal` class, defined as:

```
public static object BindToMoniker(string monikerName);
```

`BindToMoniker( )` accepts a moniker string as a parameter and returns the corresponding object. The `Marshal` class is defined in the `System.Runtime.InteropServices` namespace.

The `BindToMoniker( )` method of the `Marshal` class makes writing managed clients for a queued component as easy as if it were a COM client:

```
using System.Runtime.InteropServices;//for the Marshal
class

IMyInterface obj;
obj
=(IMyInterface)Marshal.BindToMoniker("queue:/new:MyNamesp
ace.MyComponent");
obj.MyMethod( );//call is recorded
```

In the case of a COM client, the recorder records the calls the client makes. The recorder only dispatches them to the queued component queue (more precisely, to its application's queue) when the client releases the recorder. A managed client does not use reference counting, and the recorded calls are dispatched to the queued component queue when the managed wrapper around the recorder is garbage collected. The client can expedite dispatching the calls by explicitly forcing the managed wrapper around the recorder to release it, using the static `DisposeObject( )` method of the `ServicedComponent` class, passing in the recorder object:

```
using System.Runtime.InteropServices;//for the Marshal
class

IMyInterface obj;
obj
=(IMyInterface)Marshal.BindToMoniker("queue:/new:MyNamesp
ace.MyComponent");
obj.MyMethod( );//call is recorded
```

```
//Expedite dispatching the recorded calls by disposing of
the recorder
ServicedComponent sc = obj as ServicedComponent;
If(sc !=null)
  ServicedComponent.DisposeObject(sc);
```
You can use the `IDisposable` interface instead of calling `DisposeObject()`.

10.16.3 Queued Component Error Handling

Due to the nature of an asynchronous queued call, managing a failure on both the client's side (failing to dispatch the calls) and the server's side (repeatedly failing to execute the call—a poison message) requires a special design approach. As discussed in Chapter 8, both the clients and server can use a queued component exception class to handle the error. You can also provide your product administrator with an administration utility for moving messages between the retry queues.

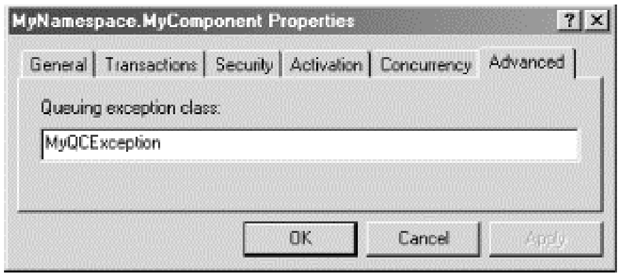
10.16.3.1 Queued component exception class

You can designate a managed class as the exception class for your queued component using the `ExceptionClass` attribute. Example 10-15 demonstrates using the `ExceptionClass` attribute.

Example 10-15. Using the ExceptionClass attribute to designate an error-handling class for your queued component

```
using COMSVCSLib;

public class MyQCException :
IPlaybackControl,IMyInterface
{
   public void FinalClientRetry(  ) {...}
   public void FinalServerRetry(  ) {...}
   public void MyMethod(  ){...}
}
[ExceptionClass("MyQCException")]
public class MyComponent :ServicedComponent,IMyInterface
{...}
```
In Example 10-15, when you register the assembly containing `MyComponent` with COM+, on the component's Advanced tab, the Queuing exception class field will contain the name of its exception class—in this case, `MyQCException`, as shown in Figure 10-3.

Figure 10-3. After registering the component in Example 10-15 with COM+, its Advanced tab contains the exception class



Queuing exception class:
MyQCException

You need to know a few more things about designating a managed class as a queued component's exception class. First, it has nothing to do with .NET error handling via exceptions. The word *exception* is overloaded. As far as .NET is concerned, a queued component's exception class is not a .NET exception class. Second, the queued component exception class has to adhere to the requirements of a queued component exception class described in Chapter 8. These requirements include implementing the same set of queued interfaces as the queued component itself and implementing the `IPlaybackControl` interface. To add `IPlaybackControl` to your class definition you need to add a reference in your project to the COM+ Services type library. `IPlaybackControl` is defined in the `COMSVCSLib` namespace.

10.16.3.2 The MessageMover class

As explained in Chapter 8, COM+ provides you with the `IMessageMover` interface, and a standard implementation of it, for moving all the messages from one retry queue to another. Managed clients can access this implementation by importing the COM+ Services type library and using the `MessageMover` class, defined in the `COMSVCSLib` namespace. Example 10-16 implements the same use-case as Example 8-2.

Example 10-16. MessageMover is used to move messages from the last retry queue to the application's queue

```
using COMSVCSLib;

IMessageMover messageMover;
int moved;//How many messages were moved

messageMover = (IMessageMover) new MessageMover(  );

//Move all the messages from the last retry queue to the
application's queue
messageMover.SourcePath = @".\PRIVATE$\MyApp_4";
messageMover.DestPath   = @".\PUBLIC$\MyApp";

moved = messageMover.MoveMessages(  );
```

10.17 COM+ Loosely Coupled Events

.NET provides managed classes with an easy way to hook up a server that fires events with client sinks. The .NET mechanism is certainly an improvement over the somewhat cumbersome COM connection point protocol, but the .NET mechanism still suffers from all the disadvantages of tightly coupled events, as explained at the beginning of Chapter 9. Fortunately, managed classes can easily take advantage of COM+ loosely coupled events.

The `EventClass` attribute is used to mark a serviced component as a COM+ event class, as shown in Example 10-17.

Example 10-17. Designating a serviced component as an event class using the EventClass attribute

```
public interface IMySink
{
   void OnEvent1( );
   void OnEvent2( );
}

[EventClass]
public class MyEventClass : ServicedComponent,IMySink
{
   public void OnEvent1( )
   {
      throw(new NotImplementedException(exception));
   }
   public void OnEvent2( )
   {
      throw(new NotImplementedException(exception));
   }
   const string exception = @"You should not call an
event class directly.
                                Register this assembly using
RegSvcs /reconfig";
}
```

The event class implements a set of sink interfaces you want to publish events on. Note that it is pointless to have any implementation of the sink interface methods in the event class, as the event class's code is never used. It is used only as a template, so that COM+ could synthesize an implementation, as explained in Chapter 9 (compare Example 10-17 with Example 9-1). This is why the code in Example 10-17 throws an exception if anybody tries to actually call the methods (maybe as a result of removing the event class from the Component Services Explorer).

When you register the assembly with COM+, the event class is added as a COM+ event class, not as a regular COM+ component.

Any managed class (not just serviced components) can publish events. Any managed class can also implement the sink's interfaces, subscribe, and receive the events. For example, to publish events using the event class from Example 10-17, a managed publisher would write:

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

The `OnEvent1( )` method returns once all subscribers have been notified, as explained in Chapter 9.

Persistent subscriptions are managed directly via the Component Services Explorer because adding a persistent subscription is a deployment-specific activity. Transient subscriptions are managed in your code, similar to COM+ transient subscribers.

The `EventClass` attribute has two public Boolean properties you can set, called `AllowInprocSubscribers` and `FireInParallel`. These two properties correspond to the Fire in parallel and Allow in-process subscribers, respectively, on the event class's Advanced tab. You can configure these values on the event class definition:

```
[EventClass(AllowInprocSubscribers =
true,FireInParallel=true)]
public class MyEventClass : ServicedComponent,IMySink
{...}
```

The `EventClass` attribute has an overloaded default constructor. If you do not specify a value for the `AllowInprocSubscribers` and `FireInParallel` properties, it sets them to `true` and `false`, respectively. Consequently, the following two statements are equivalent:

```
EventClass]
[EventClass(AllowInprocSubscribers =
true,FireInParallel=false)]
```

10.18 Summary

Throughout this book, you have learned that you should focus your development efforts on implementing business logic in your components and rely on COM+ to provide the component services and connectivity they need to operate. With .NET, Microsoft has reaffirmed its commitment to this development paradigm. From a configuration management point of view, the .NET integration with COM+ is superior to COM under Visual Studio 6.0 because .NET allows you to capture your design decisions in your code, rather than use the separate COM+ Catalog. This development is undoubtedly just the beginning of seamless support and better integration of the .NET development tools, runtime, component services, and the component administration environment. COM+ itself (see Appendix B) continues to evolve, both in features and in usability, while drawing on the new capabilities of the .NET platform. The recently added ability to expose any COM+ component as a web service is only a preview of the tighter integration of .NET and COM+ we can expect to see in the future.

Appendix A. The COM+ Logbook

One of the most effective steps you can take towards achieving a more robust application that is faster to market is adding a logging capability to your application. This appendix presents you with the COM+ Logbook, a simple utility you can implement to log method calls, events, errors, and various COM+ information. The logbook is your product's flight recorder. In a distributed COM+ environment, it is worth its weight in gold. It saved my skin whenever I tried to analyze why something did not work the way it was supposed to. By examining the log files, you can analyze what took place across machines and applications, and the source of the problem is almost immediately evident. The logbook is also useful in post-deployment scenarios to troubleshoot customer problems—just have your customer send you the log files.

A.1 Logbook Requirements

The goals for this logbook are as follows:

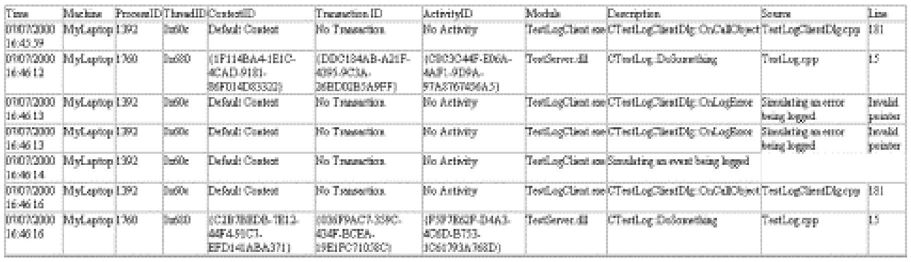
- Trace the calling tree (the causality) from the original client down to the lowest components, across threads, processes, and machines—tracing the logical thread of execution.
- Log the call's/event's/error's time and location.
- Interleave all the calls from all applications into one log file.
- Log the current COM+ execution context.
- Allow administrative customization to determine what is logged—for example, just errors, or events and errors.
- Allow administrative customization of the log filename.
- Make logging and tracing as easy as possible.
- Save log data in two formats: HTML or XML.
- Have a different lifeline for the logbook application and the applications using it.
- Be able to toggle logging on or off.

The COM+ Logbook is a COM+ server application that implements these requirements. In addition to being used by COM+ applications, it can be used in any Win32 application (such as MFC or classic COM.) The only requirement is that the application needs to run on Windows 2000.

A.2 Log File Example

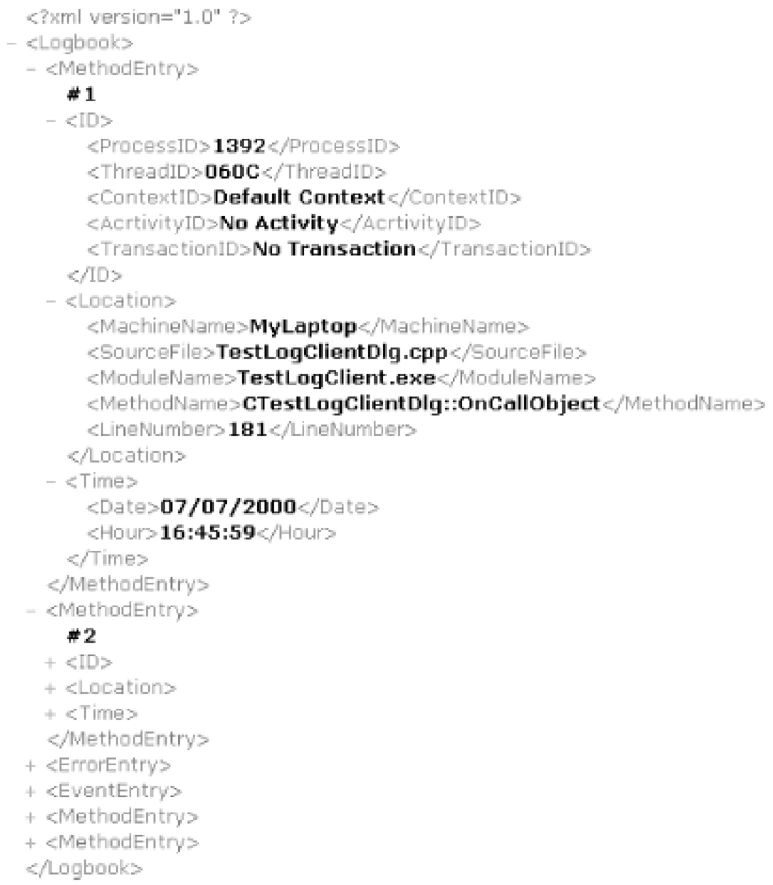
Figures A-1 and A-2 show the same tracing and logging entries—one in HTML format and the other in XML. The HTML log file is already well formatted and can be viewed by a user as is. The XML log file is less presentable.
Each entry in a log file contains the entry number (different numbers for method calls, events, and errors); the call, error, and event time; machine name; process ID; thread ID; context ID; transaction ID; activity ID; the module name (the EXE or DLL name); the method name or the error/event description; the source filename; and the line number.

Figure A-1. Logging entries in HTML



Time	Machine	ProcessID	ThreadID	ContextID	Transaction ID	ActivityID	Module	Description	Source	Line
07/07/2000 16:45:59	MyLaptop	1392	0x60c	Default Context	No Transaction	No Activity	TestLogClient.exe	CTestLogClientDlg::OnCallObject	TestLogClientDlg.cpp	181
07/07/2000 16:46:12	MyLaptop	1760	0x680	{1F114BA4-1E1C-4CAD-9181-86F014D83322}	{DDC134AB-A21F-4095-9C3A-26BD02B5A9FF}	{C8C3C44F-E06A-4AF1-9D9A-97A8767456A5}	TestServer.dll	CTestLog::DoSomething	TestLog.cpp	15
07/07/2000 16:46:13	MyLaptop	1392	0x60c	Default Context	No Transaction	No Activity	TestLogClient.exe	CTestLogClientDlg::OnLogError	Simulating an error being logged	Invalid pointer
07/07/2000 16:46:13	MyLaptop	1392	0x60c	Default Context	No Transaction	No Activity	TestLogClient.exe	CTestLogClientDlg::OnLogError	Simulating an error being logged	Invalid pointer
07/07/2000 16:46:14	MyLaptop	1392	0x60c	Default Context	No Transaction	No Activity	TestLogClient.exe	Simulating an event being logged		
07/07/2000 16:46:16	MyLaptop	1392	0x60c	Default Context	No Transaction	No Activity	TestLogClient.exe	CTestLogClientDlg::OnCallObject	TestLogClientDlg.cpp	181
07/07/2000 16:46:16	MyLaptop	1760	0x680	{C2B7B8ED8-7E12-44F4-91C7-EFD141ABA371}	{036F9AC7-339C-434F-BCEA-19E1FC71038C}	{F5F7B62F-D4A3-4C6D-B753-1C61793A768D}	TestServer.dll	CTestLog::DoSomething	TestLog.cpp	15

Figure A-2. Logging entries in XML



```
  <?xml version="1.0" ?>
- <Logbook>
  - <MethodEntry>
      #1
    - <ID>
        <ProcessID>1392</ProcessID>
        <ThreadID>060C</ThreadID>
        <ContextID>Default Context</ContextID>
        <AcrtivityID>No Activity</AcrtivityID>
        <TransactionID>No Transaction</TransactionID>
      </ID>
    - <Location>
        <MachineName>MyLaptop</MachineName>
        <SourceFile>TestLogClientDlg.cpp</SourceFile>
        <ModuleName>TestLogClient.exe</ModuleName>
        <MethodName>CTestLogClientDlg::OnCallObject</MethodName>
        <LineNumber>181</LineNumber>
      </Location>
    - <Time>
        <Date>07/07/2000</Date>
        <Hour>16:45:59</Hour>
      </Time>
    </MethodEntry>
  - <MethodEntry>
      #2
    + <ID>
    + <Location>
    + <Time>
    </MethodEntry>
  + <ErrorEntry>
  + <EventEntry>
  + <MethodEntry>
  + <MethodEntry>
  </Logbook>
```

A.3 Using the Logbook

Before using the logbook, you need to install it. Download the *logbook.msi* installation package and the header file *ComLogBook.h* from the O'Reilly web site for this book at http://www.oreilly.com/catalog/comdotnetsvs (the logbook source files and a Windows 2000 help file are also available for download). Then install the *msi* file.
After installing the logbook application, all you have to do on the side of the application doing the logging is include the *ComLogBook.h* header file in your application. The *ComLogBook.h* header file defines four helper macros for logging, described in Table A-1. Insert these macros in your code. The macros collect information on the application side and post it to the logbook.

Table A-1. The logging macros

Macro name	Description
`LOGMETHOD( )`	Traces a method call into the logbook
`LOGERROR( )`	Logs an error into the logbook
`LOGEVENT( )`	Logs an event into the logbook
`LOGERROR_AND_RETURN( )`	Logs an error into the logbook and returns in case of an error, or continues to run if no error has occurred

The macros can be used independently of one another and in every possible combination. For example, to trace a method call into the logbook, pass the method name as a string parameter to the `LOGMETHOD( )` macro:

```
void CMyClass::MyMethod(  )
{
  LOGMETHOD("CMyClass::MyMethod");
  //Real work starts here
}
```

I recommend using `LOGMETHOD( )` before doing anything else in the method body. Along with the method name, the macro logs all the required information mentioned earlier. Similarly, you can use `LOGEVENT( )` to log events and `LOGERROR( )` to log errors (see Example A-1).

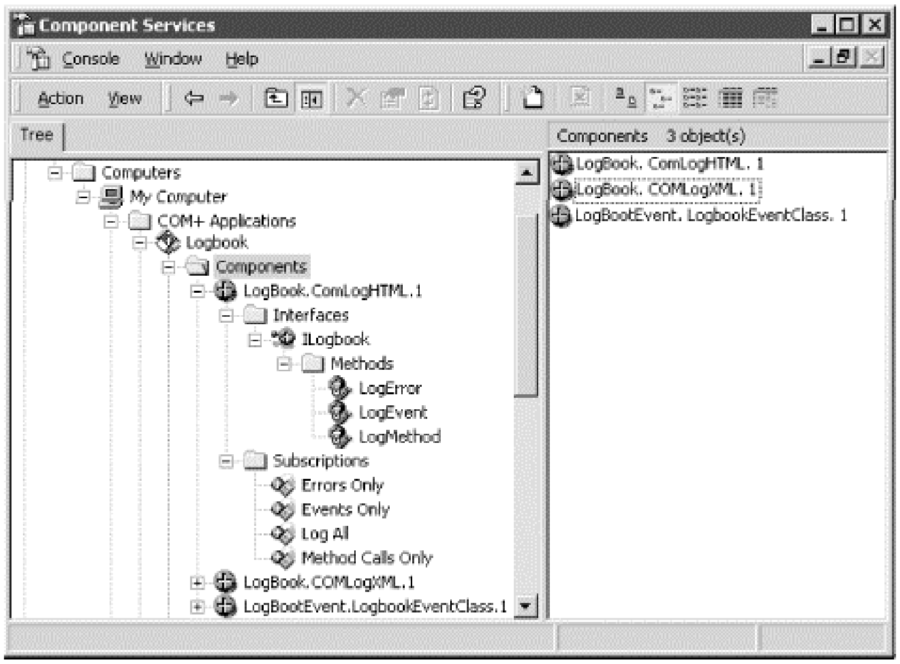
Example A-1. Using the LOGERROR( ) and the LOGEVENT( ) macros

```
//Logging an error:
void CMyClass::MyMethod(  )
{
  LOGMETHOD("CMyClass::MyMethod");
  //Real work starts here
  /*
  some code that encountered an error with a pointer
  */

LOGERROR(IID_MyInterface,E_POINTER,"CMyClass::MyMethod","
The server
                                        returned an
invalid address");
  //Continue to run
}

//logging an event into the logbook: specify in free form
text describing the event:
void CMyClass::MyMethod(  )
{
  LOGMETHOD("CMyClass::MyMethod");
  //Real work starts here
  /*
  some code that decides to log an event
  */
  LOGEVENT("The User is banging on the keyboard");
  //Continue to run
}
```

A.4 Configuring the Logbook

Configuring the various logging options is done directly via the Component Services Explorer. After installing the logbook, you will have a new COM+ application called Logbook with three components—the HTML logger, the XML logger, and an event class (see Figure A-3). All three components implement the `ILogbook` interface with the methods `LogError( )`, `LogEvent( )`, and `LogMethod( )`. The HTML and XML components have four persistent subscriptions—one for each `ILogBook` method and one for all the methods on the interface.

Figure A-3. The Logbook application has three components: the HTML logger, the XML logger, and an event class

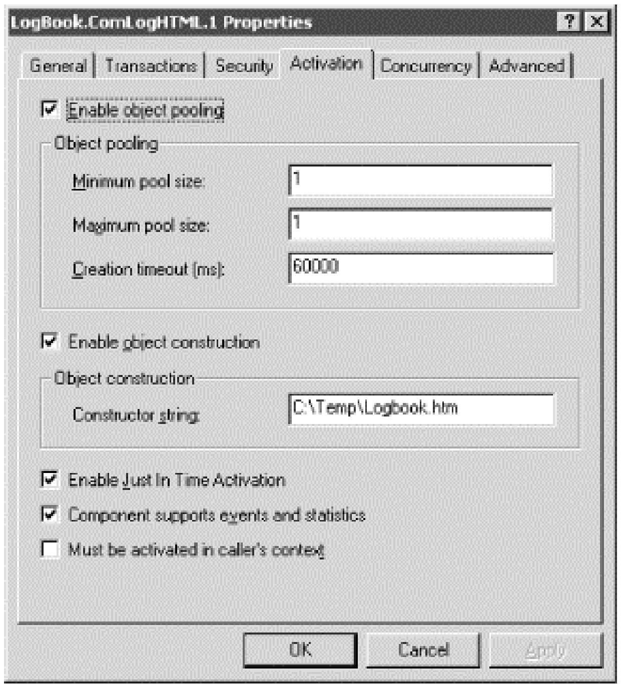


The main mechanism behind the logging is COM+ loosely coupled events. The macros publish the data as COM+ events, and the logbook components are persistent subscribers. Each logbook component has four persistent subscriptions in its Subscription folder: Errors Only, Methods Only, Events Only, and Log All.
By enabling or disabling a subscription, you can control what is being logged and in what format. After installation, by default, both the HTML and the XML Log All subscriptions are enabled and the other subscriptions are disabled. If, for example, you wish to have only HTML logging of events and errors, you should follow these steps:

- Go to the XML components, select the Log All subscription, display its properties page, go to the Options tab, and disable the subscription.
- Disable the HTML component's Log All subscription.
- Enable the HTML component's Errors Only and Events Only subscriptions.

The HTML and XML components, by default, will log to the files *C:\Temp\Logbook.htm* and *C:\Temp\Logbook.xml*, respectively. The filenames are provided as constructor strings to the components. To specify a different filename (for example, *D:\MyLog.htm* for the HTML component), display the HTML component properties and select the Activation tab (see Figure A-4). Under Object construction, specify the new filename.

Figure A-4. The logbook component properties page Activation tab



[✓] Enable object pooling

Object pooling

Minimum pool size:	1
Maximum pool size:	1
Creation timeout (ms):	60000

[✓] Enable object construction

Object construction

Constructor string:	C:\Temp\Logbook.htm

[✓] Enable Just In Time Activation

[✓] Component supports events and statistics

[ ] Must be activated in caller's context

One interesting aspect of the logbook is that its lifeline is independent from that of the applications using it because it uses persistent subscriptions. As a result, logging from many application runs will all be concatenated in the same file. If you want the logbook to start a new log file, you have to shut down the logbook application manually (right-click it in the Component Services Explorer and select Shut Down). Next time an application publishes to the logbook, the logbook clears the file and starts afresh. You can do that even when the application doing the logging is running.

A.5 How Does the Logbook Work?

The logbook uses COM+ events to pass the information collected from the application to the logbook components. The components (the HTML and XML versions) implement the `ILogbook` interface (see Figure A-3), a custom interface with methods corresponding to what is being logged—method call, event, or error. The `ILogbook` interface is defined as:

```
interface ILogbook : IUnknown
{
  typedef struct tagLOG_ENTRY
  {
    HRESULT  hres;
    DWORD    dwErrorCode;
    DWORD    dwProcessID;
    DWORD    dwThreadID;
    GUID     guidActivityID;
    GUID     guidTransactionID;
    GUID     guidContextID;
    BSTR     bstrMachineName;
    BSTR     bstrSourceFileName;
    BSTR     bstrModuleName;
    BSTR     bstrMethodName;
    DWORD    dwLineNumber;
    BSTR     bstrDescription;
    IID      iidError;
    FILETIME eventTime;
  }LOG_ENTRY;

  HRESULT LogError ([in]LOG_ENTRY* pErrorEntry);
  HRESULT LogMethod([in]LOG_ENTRY* pMethodEntry);
  HRESULT LogEvent ([in]LOG_ENTRY* pEventEntry);
};
```

The helper macros collect the information on the application side, pack it into a `LOG_ENTRY` struct, create a COM+ event class that implements `ILogbook`, and fire the appropriate event. The logbook receives the event, formats it appropriately (to HTML or XML), and writes it to the log file.

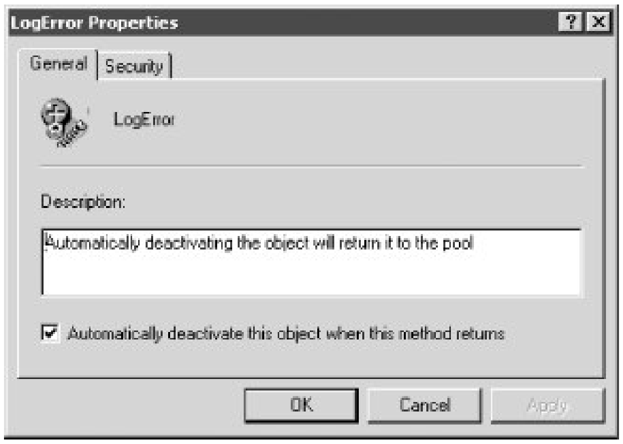
Deciding to use COM+ events was the easy part of the design. Deciding how to channel all the events to the same logbook component and how to collect all the tracing information you are interested in is more challenging.

To solve the first challenge, you can use COM+ instance managements services. The components in the logbook application are configured to use Object Pooling and Just-in-Time Activation (JITA) to create the COM+ equivalent of a singleton (as discussed in Chapter 3). Each component (HTML and XML) implements the `IObjectControl` interface and returns `TRUE` from `IObjectControl::CanBePooled( )`. The object pool is configured to have a minimum and a maximum pool size of 1, ensuring that there is always exactly one instance of a component of that type (see Figure A-4).

When a logging client application publishes an event, COM+ delivers the event to the persistent subscriptions of the logbook component. But because the logbook component is pooled, with a pool size of exactly 1, COM+ does not create a new instance of the persistent subscriber. Instead, it retrieves the logbook component from the pool, hands the event to the component, and releases it back to the pool once the method returns. However, what would happen if a greedy application created the logbook component directly and held on to it? The maximum pool size is 1, so COM+ wouldn't create another instance of the logbook component to publish the event to it, but would instead wait for the existing object to return to the pool. The object wouldn't return, though, since the greedy application would be holding a reference to it. As a result, all attempts from other applications to publish to the logbook would fail after the timeout specified in the Creation Timeout (see Figure A-4). As you saw in Chapter 3, JITA is designed to handle such greedy clients. If the logbook component indicates to COM+ that it is willing to be deactivated and is configured to use JITA, COM+ deactivates the component. In this case, it returns back to the pool, as opposed to a real release. The greedy application does not know the difference because it still has a reference to a valid proxy. The next time the greedy client application tries to access the logbook, COM+ detects it, retrieves the object from the pool, and hooks it up with the interceptor so the greedy application's logging call goes through.

The logbook components are therefore configured to use JITA (see Figure A-4). However, a logbook component still has to let COM+ know when it is okay to deactivate it. The logical place would be at method boundaries when it is done logging to the file. Therefore, the logbook components use COM+ method auto-deactivation (see Figure A-5). Every logging method is configured to automatically deactivate the object on return.

Figure A-5. COM+ deactivates the object after each method call



Description: Automatically deactivating the object will return it to the pool

☑ Automatically deactivate this object when this method returns

Because the logbook application is a server application, there is little impact for the component's threading model, since all calls are marshaled across process boundaries anyway. For the remote possibility of ever being deployed in a library application, the logbook components use the `Both` threading model. Synchronization is provided by having the components' synchronization configured as Required. Note that, as explained in Chapter 5, JITA requires synchronization, so the only available synchronization settings are Required and Requires New.

One other configuration setting used is to have COM+ leave the logbook application running when idle (on the Logbook application properties page, Advanced tab). This is required to keep the pool alive, even if there are no external clients using logging. As a result, all logging is written to the same file. Because you only have to create a new application and component instance once, performance improves.

You already saw that the filename is passed as a constructor string. As explained in Chapter 3, the logbook components implement IObjectConstruct to access that string. COM+ queries for that interface after creating the object. Then it passes to the only method, Construct( ), a pointer to an IObjectConstructString object. You can use that pointer to get the constructor string, which is a filename in this case. Look at Example 3-2 in Chapter 3 to learn how to gain access to the constructor string.

The other major challenge in developing the logbook is collecting the information on the client side. Some of it, like line number, file, and module name, has nothing to do with COM+ and are just neat programming tricks that use predefined compiler macros; look at the source files if you are curious. Obtaining the execution and context IDs is another thing. Fortunately, COM+ has an excellent infrastructure for this purpose: the IObjectContextInfo interface. As demonstrated in Chapter 2 and and other chapters in the book, you can use the IObjectContextInfo interface to retrieve the context, transaction, and activity ID. This is exactly what the helper macros (Table A-1) do on the client side. The macros actually use a helper class, CEventLogger, to collect the information and publish it to the logbook. Example A-2 shows how the LOGEVENT macro is implemented.

Example A-2. The LOGEVENT helper macro

```
#define LOGEVENT(x)  DoLogEvent(x)

inline void DoLogEvent(const CString& sEvent)
{
   CEventLogger eventLogger;
   eventLogger.DoLogEvent(sEvent);
}

inline void CEventLogger::DoLogEvent(const CString&
sEvent) const
{
   LOG_ENTRY logEntry;
   HRESULT    hres = S_OK;
   ILogbook* pLogbook   = NULL;

   FillLogEntry(&logEntry,sEvent);//using
IObjectContextInfo to get the IDs

   //Create the event class and publish
   hres = ::CoCreateInstance(CLSID_LogbookEventClass,..,
                             IID_ILogbook,&pLogbook);
   //Publish to the logbook
   hres = pLogbook->LogEvent(&logEntry);
   pLogbook->Release(  );
}
```

A.6 Summary

The logbook makes elegant use of many COM+ features, such as the event system, Just-in-Time Activation (JITA), object pooling, idle time management, automatic deactivation of objects, synchronization, and the object constructor string. The logbook is a good example of the synergies generated by using multiple services simultaneously. You can extend the logbook or improve it by customizing it to fit specific requirements (such as adding verbosity levels). In any case, once you start enjoying the productivity boost of the logbook, you will find yourself asking one question: "How did I ever manage without it?"

Appendix B. COM+ 1.5

The next release of Windows 2000, *Windows XP,* will be the first Windows operating system to include the next version of COM+, called COM+ 1.5. This appendix describes the new features and capabilities of this future release of COM+. The current version of COM+ is referred to as COM+ 1.0.

In COM+ 1.5, Microsoft improved COM+ usability in a number of ways and addressed some of COM+ 1.0's pitfalls described in this book. Microsoft also added new features to existing services and laid the foundation for integration with .NET services. COM+ 1.5 is fully backward-compatible with COM+ 1.0 components and applications. In fact, when you export a COM+ 1.5 application, the export wizard lets you export the application in COM+ 1.0 format to be installed on machines running COM+ 1.0 (although the new features and properties will be lost in such an export).

The COM+ Catalog interfaces and collections have been extended to handle the new additions. When describing a new service, the new corresponding Catalog items are provided whenever possible because no other public documentation is currently available.

B.1 Improved User Interface Usability

Under COM+ 1.0, the only way to know the activation type of a COM+ application was to bring up its Activation tab and examine it. The COM+ 1.5 Explorer assigns different icons to different application types, so you can deduce the application's type (server, library, or proxy) just by viewing it. Service applications, (discussed shortly), a fourth application type available in COM+ 1.5, also have a distinct icon. A new folder under My Computer called *Running Processes* contains all the currently executing applications for easy runtime administration.

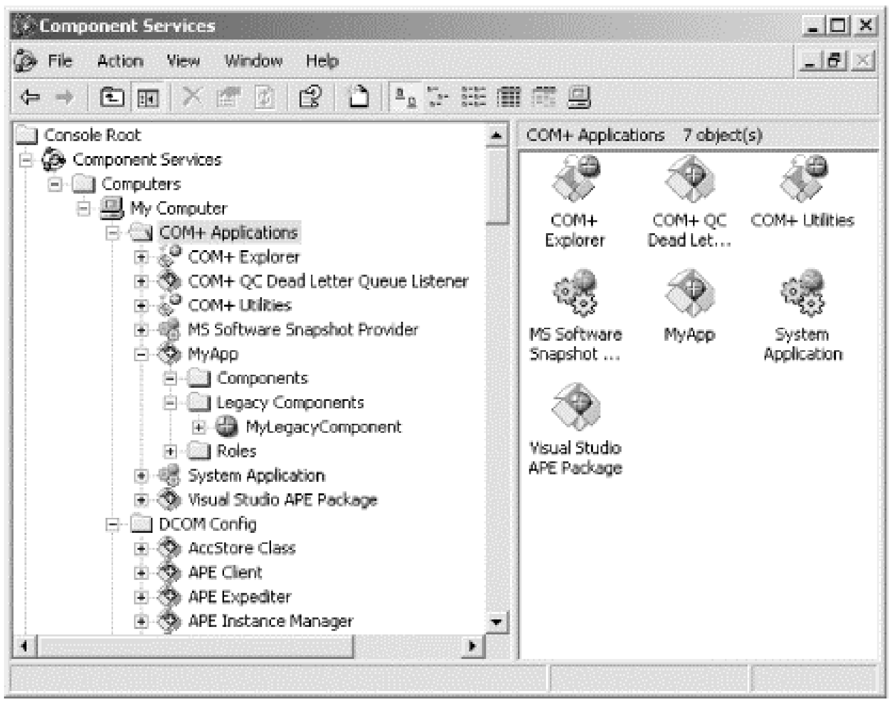
B.2 Legacy Applications and Components

The COM+ 1.0 Explorer only allows you to manage configured components. If your product is made up entirely of configured components, then that limitation is probably fine to you. However, not all developers are that lucky. In real life, configured components often have to interact with in-house or third-party legacy COM components. In such a heterogeneous environment, developers use such tools as DCOMCNFG, OLEView, Visual Studio, or custom tools to manage legacy components in addition to the Component Services Explorer. Developers also have to manage two types of deployment approaches—one that uses exported COM+ applications (MSI files) and another that is whatever they need to install the legacy components. One new feature of COM+ 1.5 is complete support for legacy applications and components, which allows you to manage every aspect of your legacy applications and components just as well as DCOMCNFG and OLEView do.

B.2.1 Legacy Applications

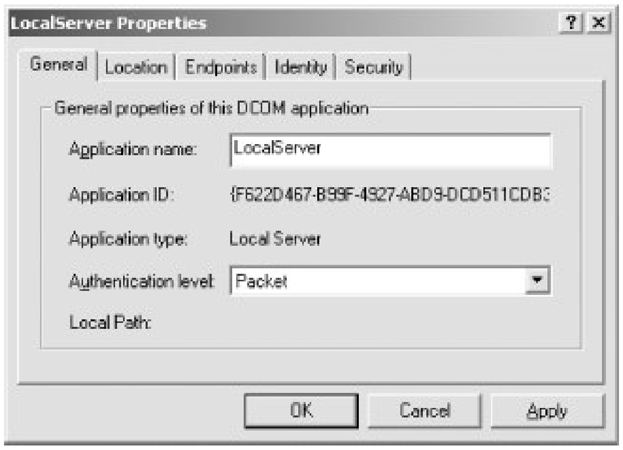
In the COM+ 1.5 Explorer, under the My Computer icon, there is a new folder called DCOM Config. This folder is a sibling to the *COM+ Applications* folder (see Figure B-1).

Figure B-1. The COM+ 1.5 Explorer



The *DCOM Config* folder contains all the registered COM local servers (EXE servers) on your machine. Each local server is called a *legacy application.* Unlike a COM+ application, you cannot expand a legacy application down to the component, interface, or method level. A legacy application is opaque as far as COM+ 1.5 is concerned. The DCOM Config folder simply gives you a centralized place to manage both your COM+ applications and your legacy local servers without resorting to other utilities. When you right-click on a legacy application and select Properties from the pop-up context menu, you get a properties page that lets you manage every aspect of the legacy application, much like what DCOMCNFG provides (see Figure B-2).

Figure B-2. The properties page of a legacy application



General properties of this DCOM application

Application name: LocalServer

Application ID: {F622D467-B99F-4927-ABD9-DCD511CDB3

Application type: Local Server

Authentication level: Packet

Local Path:

The General tab lets you change the application name and set the authentication level for incoming calls to this application. The Location tab lets you control whether to run the application on your computer or on another computer on the network. The Endpoints tab lets you configure the transport protocols for the DCOM calls. The Identity tab lets you specify under which security identity to launch the server, including the system account (for services only). The Security tab lets you configure users' access, launch, and change permissions.

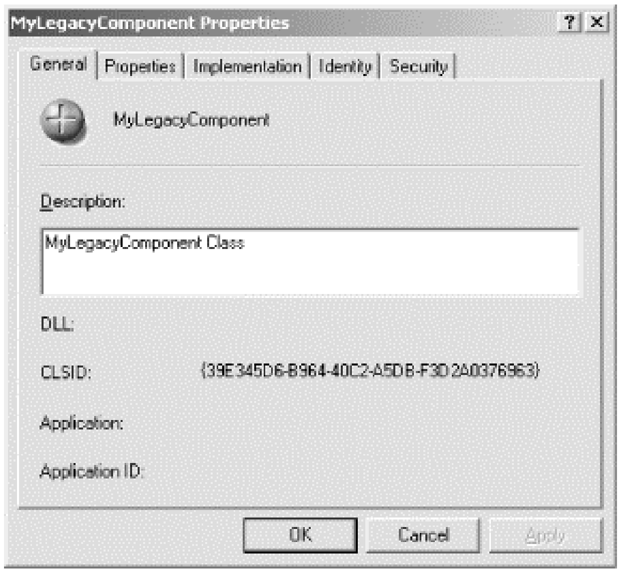
COM+ 1.5 defines a new top-level catalog collection called `LegacyServers`. Every catalog object in that collection corresponds to a local server and provides the main Registry entries (`CLSID`, `ProgID`, `ClassName`, `LocalServer32`, and `InprocServer32`) as named properties.

B.2.2 Legacy Components

COM+ 1.5 calls nonconfigured in-proc COM components *legacy components.* If your COM+ application uses legacy components, the COM+ 1.5 Explorer lets you manage them within the scope of your application as well. Every COM+ 1.5 application has a new folder called *Legacy Components* (see Figure B-1). The *Legacy Components* folder is a sibling to the *Components* folder. To add a legacy component to the folder, expand it, right-click on it, and select New from the context menu. The COM+ 1.5 Explorer brings up the Legacy Component Import Wizard. The wizard lets you choose legacy components (registered in-proc components) to add to your application. Like configured components, legacy components can take part in at most one COM+ 1.5 application. The major benefit of having your legacy components as part of your COM+ 1.5 application is deployment. When you export a COM+ application, its MSI file contains the legacy components and their settings. When you install the MSI file on another machine, the Windows Installer registers the components, thus saving you the trouble of writing a separate installation program.

The properties page of a legacy component presents you with every relevant Registry entry for that component (see Figure B-3).

Figure B-3. The properties page of a legacy component



MyLegacyComponent

Description:

MyLegacyComponent Class

DLL:

CLSID: {39E345D6-B964-40C2-A5DB-F3D2A0376963}

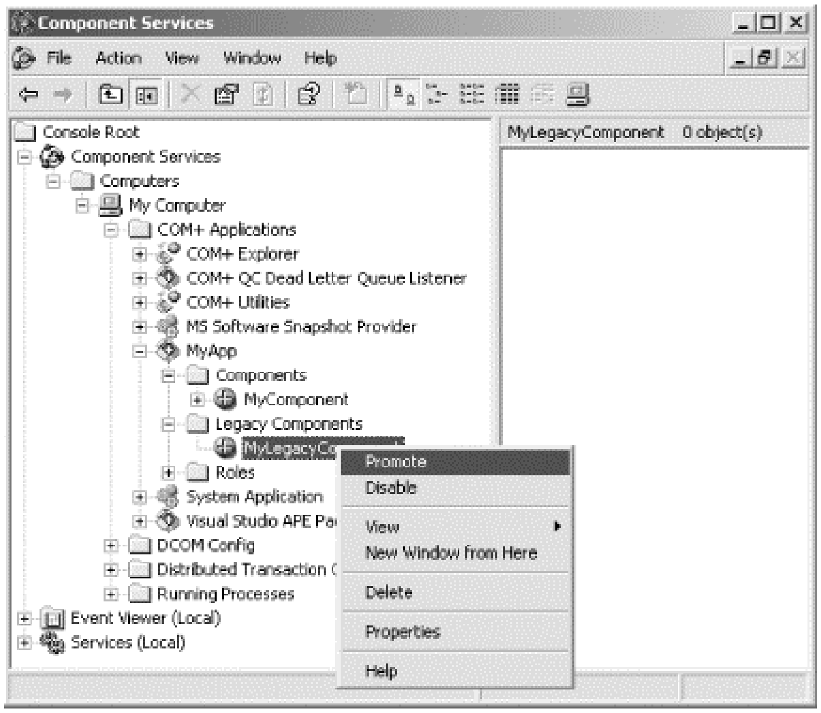
Application:

Application ID:

You can change only the values of settings that do not collide with registry settings in the component itself. For example, you cannot change the threading model value, but you can provide the name of a surrogate process.

You can even *promote* a legacy component to a configured component. Simply bring up the legacy component's context menu and select Promote (see Figure B-4). The legacy component is removed from the *Legacy Components* folder and added to the *Components* folder in the same COM+ 1.5 application.

Figure B-4. A legacy component pop-up context menu



The COM+ 1.5 Catalog root object (`COMAdminCatalog`) supports a new interface called `ICOMAdminCatalog2`, which derives from `ICOMAdminCatalog`. `ICOMAdminCatalog2` contains the following methods for handling legacy components:

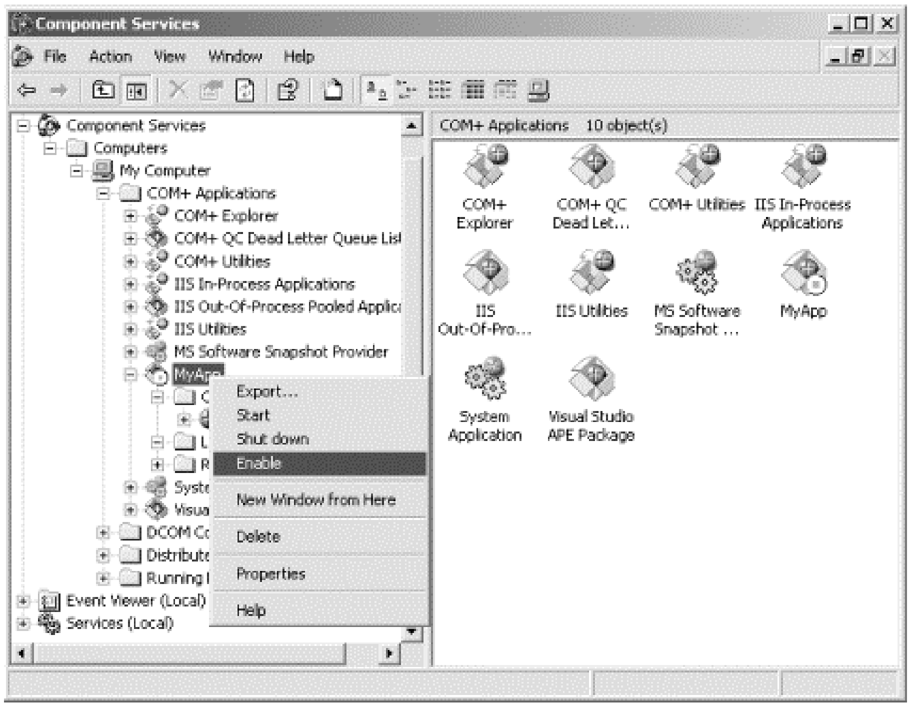
```
[id(0x2b)] HRESULT ImportComponentAsLegacy([in]BSTR
bstrAppIdOrName,
                                           [in]BSTR
bstrCLSIDOrProgId,
                                           [in]long
lComponentType);

[id(0x2c)] HRESULT PromoteLegacyComponent([in] BSTR
bstrAppIdOrName,
                                          [in] BSTR
bstrCLSIDOrProgId);
```

`ImportComponentAsLegacy( )` adds a legacy component to the specified application and `PromoteLegacyComponent( )` promotes an already imported legacy component to a configured component. In addition, every application in the COM+ 1.5 Catalog has a `LegacyComponents` collection. You can traverse this collection programmatically and configure it.

B.3 Disabling Applications and Components

The COM+ 1.5 Explorer lets you disable applications and components. When you *disable* an application, all client attempts to create any component from that application fail, and the following message is associated with the HRESULT: "The component has been disabled." To disable an application, display its pop-up context menu and select Disable. A disabled application has a red square on it (like a player's Stop button) in the COM+ 1.5 Explorer (see Figure B-5). To enable a disabled application, bring up the context menu again and select Enable. You can only disable a COM+ 1.5 application. Legacy applications cannot be disabled. Interestingly, a client that already has a reference to a COM+ object is not affected by the disabled application. Only clients that try to create new objects are affected. Consequently, you can have a disabled application running indefinitely.

Figure B-5. Disabling or enabling a COM+ 1.5 application from its pop-up context menu



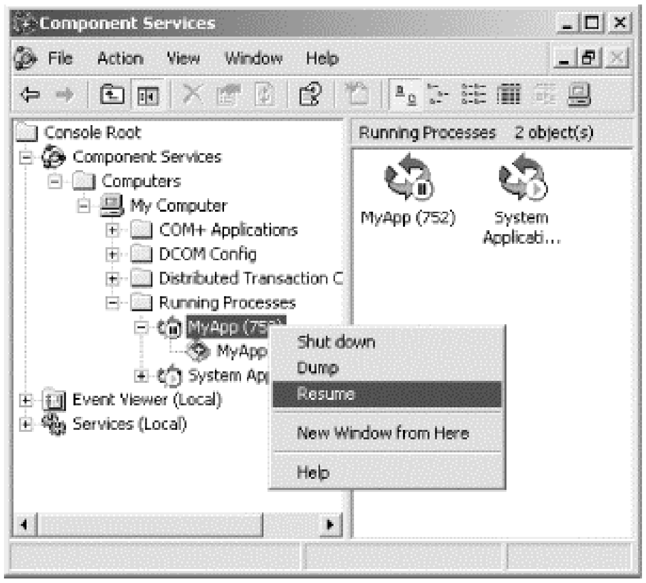
You can also disable on a component-by-component basis instead of disabling an entire application. Every component pop-up context menu has a Disable option. Like a disabled application, a disabled component has a red square on it. All client attempts to create a disabled component fail, and the following message is associated with the HRESULT: "The component has been disabled." You can disable any component in a COM+ 1.5 application, including legacy components (see Figure B-4). To enable a component, select Enable from its context menu. Like a disabled application, a disabled component only affects new activation requests. Existing references to objects are not affected. Enabled status for applications and components is stored in the COM+ Catalog and is therefore maintained between machine reboots.

Disabling applications or components is useful in two cases. The first situation is when you want to gracefully shut down an application on a live server machine to perform maintenance or upgrades. If you simply shut down the application, you might cause failures on client machines holding existing references. By disabling an application, you can have existing clients finish their work, while new activations may be routed to another machine, providing you the opportunity to perform maintenance. The other situation in which disabling an application is useful is during development and testing. It provides a guaranteed way to fail client calls and is thus a way to test your client-side error handling.
Currently, the COM+ Catalog interface `ICOMAdminCatalog2` does not have methods used to programmatically disable or enable an application, but that situation could change by release time. Another possibility is that every COM+ application catalog object will have an `Enabled` named property. Currently, an application object has a Boolean property called `IsEnabled` that is set to `TRUE` if the application is enabled and `FALSE` if it is disabled. Similarity, components today do not have an `Enabled` named property, only a Boolean property called `IsEnabled`, used the same way as in the application object.

B.4 Pausing Applications

Pausing an application is similar to disabling an application, except it is used to disable a particular running application only, and the paused status does not survive an application shutdown. To pause a running application, open the *Running Processes* folder and select Pause from the application context menu. A paused application has a paused icon on it (like a player's Pause button), while a running application has a play icon (like a player's Play button). To resume a paused application, select Resume from its context menu (see Figure B-6).

Figure B-6. PCOM+ 1.5 running application Activation tab



To pause an application programmatically, use the `ICOMAdminCatalog2` interface and the `PauseProcess( )` method. The `ResumeProcess( )` method is used to resume the application, and the `IsProcessPaused( )` method allows you to find out the status of the application. The definitions of these methods follow:

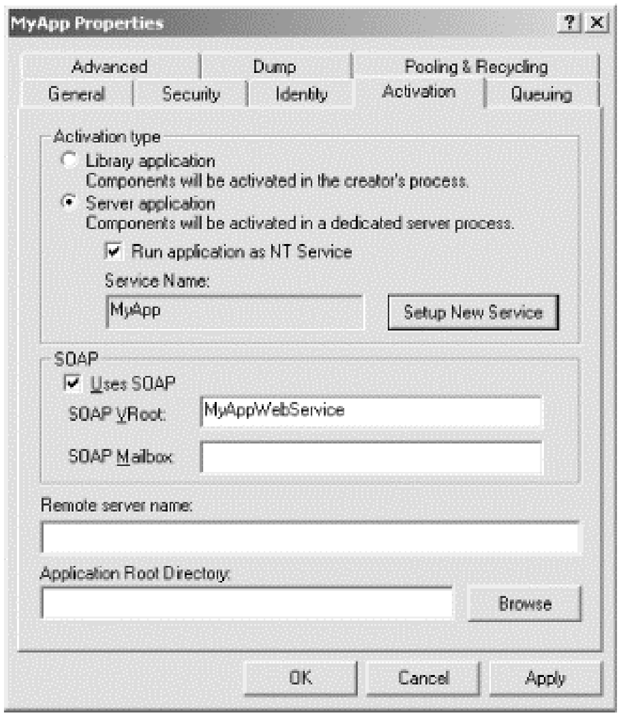
```
[id(0x1c)] HRESULT PauseProcess([in] BSTR
bstrApplInstanceId);
[id(0x1d)] HRESULT ResumeProcess([in] BSTR
bstrApplInstanceId);
[id(0x1e)] HRESULT IsProcessPaused([in] BSTR
bstrApplInstanceId,

[out,retval]VARIANT_BOOL* bPaused);
```

B.5 Service Activation Type

COM+ 1.5 allows you to configure a server application to run as a system service. Configuring your application as a service allows you to have your application running as soon as the machine boots, independent of client activation requests. Another benefit is that a service application is the only way to run under the system identity account. The system account is the most powerful account on a given machine.

The application Activation tab contains the checkbox "Run application as NT Service" (see Figure B-7). When this option is selected, you can also configure the various service parameters by clicking the Setup New Service button, saving you the trouble of using the Control Panel services applet.

Figure B-7. COM+ 1.5 application Activation tab



Activation type
( ) Library application
Components will be activated in the creator's process.
(•) Server application
Components will be activated in a dedicated server process.
[✓] Run application as NT Service
Service Name: MyApp

SOAP
[✓] Uses SOAP
SOAP VRoot: MyAppWebService
SOAP Mailbox:

Remote server name:

Application Root Directory:

`ICOMAdminCatalog2` provides you with programmatic ability to configure a service with the `CreateServiceForApplication( )` method and to unconfigure a server application as a system service with the `DeleteServiceForApplication( )` method. The service name is available through the `ServiceName` property of the application's catalog object.

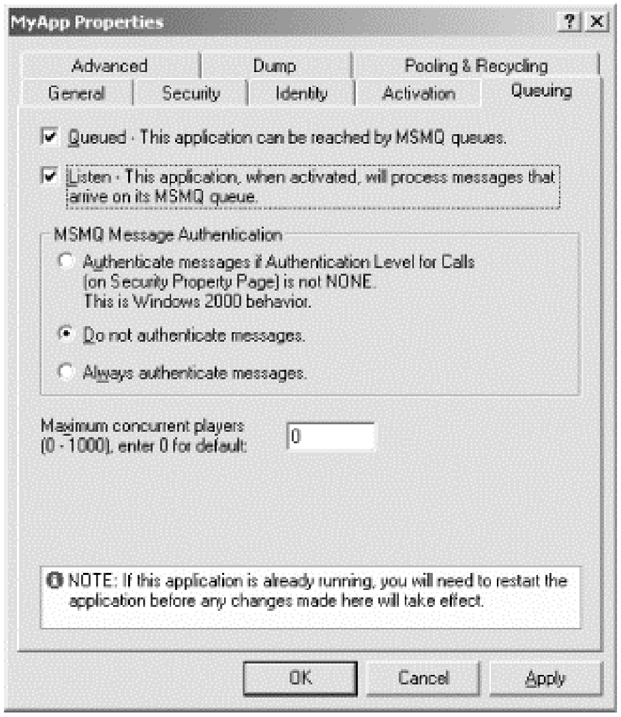
B.6 Improved Queuing Support

As explained in Chapter 8, queued components under COM+ 1.0 require the presence of a domain controller to provide authentication for the queued call. If you do not have a domain controller, you must turn off COM+ 1.0 application authentication (set it to None). COM+ 1.5 provides better configurable support for queued calls by separating them from normal synchronous calls. The application Queuing tab now lets you configure authentication for queued calls explicitly (see Figure B-8). Your available options are to:

- Use MSMQ domain controller authentication when the application is configured to use authentication for synchronous calls (when the application authentication is set to any value except None).
- Never authenticate queued calls into this application. Choosing this option allows you to use queued components freely without a domain controller.

- Always authenticate incoming queued calls, regardless of the application authentication setting.

Figure B-8. COM+ 1.5 server application Queuing tab



[✓] Queued - This application can be reached by MSMQ queues.

[✓] Listen - This application, when activated, will process messages that arrive on its MSMQ queue.

MSMQ Message Authentication

( ) Authenticate messages if Authentication Level for Calls (on Security Property Page) is not NONE. This is Windows 2000 behavior.

(●) Do not authenticate messages.

( ) Always authenticate messages.

Maximum concurrent players (0 - 1000), enter 0 for default: 0

NOTE: If this application is already running, you will need to restart the application before any changes made here will take effect.

The Queuing tab also allows you to control the maximum number of concurrent players the application can contain. Because every player is created on a separate thread, some overhead is associated with creating and maintaining a player. In extreme situations, your application may grind to a halt if the number of concurrent players is too large (a few hundred). When you set a limit on the number of players and that limit is reached, the listener does not create new players. Rather, queued calls remain in the application queue, allowing calls in progress to execute and complete. The limit is also good for load-balancing purposes and can be used in conjunction with application pooling, discussed next.

The COM+ 1.5 Catalog lets you configure the queuing support programmatically as named properties of the application catalog object. The authentication level is accessible via the `QCAuthenticateMsgs` named property, and the maximum number of players is accessible via the `QCListenerMaxThreads` property.

B.7 Application Pooling and Recycling

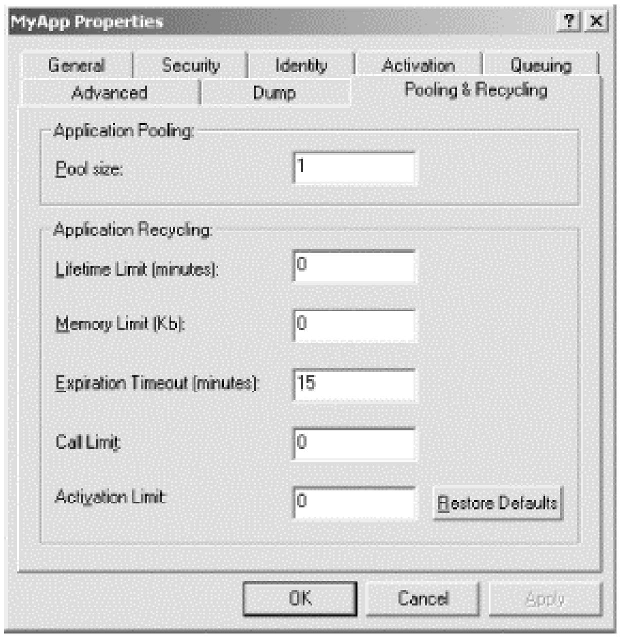
COM+ 1.5 provides two new application lifecycle management options: application pooling and recycling. Both options are configurable on a new tab on the application's properties page. Pooling and recycling services are available only for a server application. You cannot configure library applications to use pooling and recycling because they do not own their hosting process. Library applications have, in effect, the pooling and recycling parameters of whatever server application happens to load them.

B.7.1 Application pooling

Application pooling allows you to configure how many surrogate processes are launched to host your server application's components. Under COM+ 1.0, all instances of components from a server application always share the same hosting process. Although this sharing is also the classic COM default, classic COM local server developers had the option of allocating a process per object (by registering the class factories with the `REGCLS_SINGLEUSE` flag). COM+ 1.5 gives you explicit control over how many processes are launched by configuring a processes pool. The application properties page now contains the Pooling & Recycling tab (see Figure B-9). You can configure the pool size in the Pool size edit box. The default pool size is one—a single process hosts all instances of components from the application, like in COM+ 1.0. However, if you set it to a value greater than one, COM+ 1.5 creates a process per each new instance until it reaches the pool size, at which point COM+ starts multiplexing new instances to the existing processes, apparently in a round-robin fashion. The maximum configurable pool size is 999,999, enough for all practical purposes. Application pooling is useful as a fault isolation technique. If one process has to shut down because of some error or exception, the other processes and their clients are not affected. Application pooling also gives you same-machine load balancing—you do not have to use a separate load balancing service with multiple machines to allocate different instances to different processes. The pool size is available as the `ConcurrentApps` named property of an application catalog object.

> If you start a COM+ 1.5 application manually or programmatically, COM+ 1.5 creates as many processes as the configured pool size. This behavior is analogous to component minimum pool size, discussed in Chapter 3, and it only comes into play when the application is started explicitly. This behavior is useful when you want to mitigate anticipated spikes in client requests—you shouldn't pay the overhead of creating new processes (and potentially, pools of objects in those processes as well).

Figure B-9. COM+ 1.5 provides server applications with pooling and recycling services


MyApp Properties

Application Pooling:

Pool size: 1

Application Recycling:

Lifetime Limit (minutes): 0

Memory Limit (Kb): 0

Expiration Timeout (minutes): 15

Call Limit: 0

Activation Limit: 0

B.7.2 Application Recycling

The other new application lifetime management service is *recycling*. Application recycling is used to increase overall application robustness and availability by compensating for code defects. For example, one of the most common defects is a memory leak. Not all products have the necessary quality assurance resources or commitment during development; as a result, memory leaks can be present in the released product. An issue arises when a COM+ application can be left running indefinitely servicing clients. Even a very small memory leak can have a devastating effect over a long period of time. For example, imagine a system with an "insignificant" memory leak of only 10 bytes per method call. A modern system that processes in excess of 100 transactions per second will, after one day, leak 100 MB of memory. The process hosting the application will consume this amount of memory, thus severely hampering performance and availability, as a result of additional page faults and memory allocation penalties. The way developers treated such a leak in COM+ 1.0 (other than fixing it) was by periodically terminating the hosting process and restarting it. This technique is called *application recycling*. COM+ 1.5 allows you to configure automatic recycling on the Pooling and Recycling tab (see Figure B-9). You can have COM+ recycle the process when it reaches a memory limit (the Memory Limit edit box) to cope with memory leaks. A value of zero is the default value, which means no limit.

By specifying the Lifetime Limit value, you can also instruct COM+ to shut down your application after a predetermined amount of time. This instruction allows you to cope with defects in handling other kinds of resources (such as system handles) by specifying the Lifetime Limit value. A value of zero is the default value, which means no lifetime limit. Note that the semantics of the lifetime limit is different from the idle time management option on the application Advanced tab. The Server Process Shutdown value on the Advanced tab specifies after how many minutes of idle time (i.e., not servicing clients) to shut down the application. The lifetime value specifies after how many minutes to shut down the application, irrespective of the work in progress inside the process.

COM+ provides two more recycling triggers. You can have COM+ recycle your application after a specified number of method calls into your application by specifying such a limit in the Call Limit edit box. The number of calls is defined as combined number of calls made on all objects in the application. The default value is set to zero—no limit. You can also request application recycling after a certain number of activations. *Activations* is defined as the total number of objects that COM+ 1.5 created in that application. You specify the activation limit in the Activation Limit edit box and, again, the default value is set to zero.

Regardless of how the decision to recycle the application is made (the memory limit reached, the lifetime elapsed, or the call or activation limit was reached), COM+ 1.5 routes new activation requests to a new host process and waits for existing clients to release their references to objects in the recycled process. However, you can specify how long COM+ 1.5 should wait in the Expiration Timeout edit box. After that expiration timeout, COM+ 1.5 terminates the application, even if clients are still holding live references. The default expiration timeout is 15 minutes.

Finally, note that recycling is not available for a COM+ application configured as system service, nor can you recycle a paused application.

B.7.3 Programmatic Recycling

The COM+ 1.5 Catalog provides you with programmatic ability to configure the recycling parameters discussed previously. To configure memory and time-bound recycling, use the `RecycleMemoryLimit` and `RecycleLifetimeLimit` named properties of the application's catalog object. To configure the expiration timeout, use the `RecycleExpirationTimeout` named property. To configure call or activation limit programmatically, set the values of the `RecycleCallLimit` or `RecycleActivationLimit` named properties.

Example B-1 shows how to set a recycling limit programmatically. It implements the `SetRecycleByActivations( )` helper function, which sets a specified limit of activations for recycling a specified application.

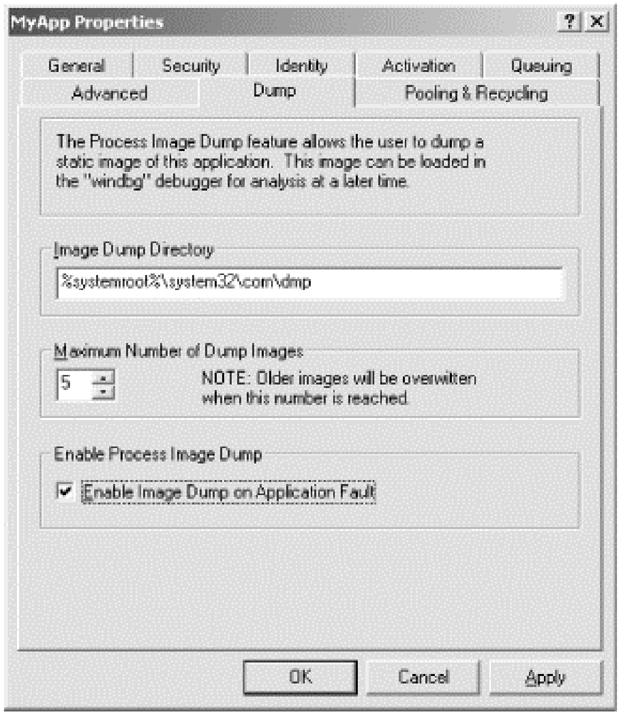
Example B-1. Setting a recycling limit programmatically

```
//usage: "MyApp" will be recycled after 1000 object
activations
//hres = SetRecycleByActivations("MyApp",1000);

HRESULT SetRecycleByActivations(LPCSTR lpcszAppName,DWORD
dwActivations)
{
   //Verify app name is valid
   if(_bstr_t(lpcszAppName) == _bstr_t(""))
   {
      return  E_INVALIDARG;
   }
   HRESULT hres = S_OK;
   ICOMAdminCatalog2* pCatalog = NULL;
   hres = ::CoCreateInstance(CLSID_COMAdminCatalog,
NULL,CLSCTX_SERVER,

IID_ICOMAdminCatalog2,(void**)&pCatalog);

   ICatalogObject* pApplication  = NULL;
   ICatalogCollection* pApplicationCollection = NULL;
   long nApplicationCount = 0;
   int i = 0;//Application index

   //Get the application collection
   hres = pCatalog-
>GetCollection(_bstr_t("Applications"),

(IDispatch**)&pApplicationCollection);
   pCatalog->Release(  );

   hres = pApplicationCollection->Populate(  );

   hres = pApplicationCollection-
>get_Count(&nApplicationCount);

   hres = COMADMIN_E_OBJECT_DOES_NOT_EXIST;
   for(i=0;i<nApplicationCount;i++)
   {
      //Get the current application
      hres = pApplicationCollection-
>get_Item(i,(IDispatch**)&pApplication);

      _variant_t varName;
      pApplication->get_Name(&varName);
      _bstr_t bstrName(varName);

      if(bstrName == _bstr_t(lpcszAppName))
      {
         long ret = 0;
         _variant_t
varActivationLimit((long)dwActivations);
         hres = pApplication-
>put_Value(_bstr_t("RecycleActivationLimit"),

varActivationLimit);
         hres = pApplicationCollection-
>SaveChanges(&ret);
      }
      pApplication->Release(  );
   }
   pApplicationCollection->Release(  );
   return hres;

}
```

B.8 Application Dump

For debug and analysis purposes, getting a complete memory dump of an application is sometimes useful, especially at the time of a crash. COM+ 1.5 allows you to configure a dump of a static memory image of the process hosting your COM+ application. You can use a utility such as WinDbg to view and analyze this image. Every COM+ 1.5 application (including library applications) has a new tab on its properties page called Dump (see Figure B-10). You can specify a location for the image dump and how many images to store there. When the maximum number of images is reached, a new dump image overwrites the oldest one. The maximum number of images you can have COM+ 1.5 store for you is 200. You can generate a dump file in several ways. The first (and most useful) way is to instruct COM+ to dump a memory image on application fault (at the bottom of the Dump tab—see Figure B-10). In this context, an application's fault is when an exception is thrown.

Figure B-10. COM+ 1.5 application Dump tab



The Process Image Dump feature allows the user to dump a static image of this application. This image can be loaded in the "windbg" debugger for analysis at a later time.

Image Dump Directory: %systemroot%\system32\com\dmp

Maximum Number of Dump Images: 5

NOTE: Older images will be overwritten when this number is reached.

Enable Process Image Dump

☑ Enable Image Dump on Application Fault

The second way to generate a dump file is to select Dump from a running application context menu (see Figure B-6). Finally, you can also request a dump explicitly by using the `DumpProcess( )` method of the `ICOMAdminCatalog2` interface, defined as:

```
[id(0x1f)] HRESULT DumpProcess([in] BSTR
bstrApplInstanceId,
                               [in] BSTR bstrDirectory,
                               [in] long lMaxImages,
                               [out,retval] BSTR*
pbstrDumpFile);
```

When you use the `DumpProcess( )` method, you have to provide the dump directory and filename and you cannot rely on the configured values. Requesting a dump (either by calling `DumpProcess( )` or selecting Dump from the context menu) on a running application is nonintrusive—the process can continue to run and is only frozen temporarily for the duration of the dump.

When COM+ generates a dumped file, it uses the following naming convention as a filename:

```
{<
App-ID>}_year_month_day_hour_minute_second.dmp
```

This convention lets you easily associate a dump file with a reported system failure. For example, here is a typical dump filename:

```
{02d4b3f1-fd88-11d1-960d-
00805fc79235}_2001_06_14_13_28_51.dmp
```

To avoid calling `DumpProcess( )` needlessly, `ICOMAdminCatalog2` has a helper method called `IsProcessDumpSupported( )`, used to find out whether image dump is enabled on the machine:

```
[id(0x20)]HRESULT
IsProcessDumpSupported([out,retval]VARIANT_BOOL*
pbSupported);
```

You can set the various dump properties programmatically as well, using named properties of the application catalog object. The `DumpEnabled` property lets you enable or disable image dump for the application, `DumpOnException` lets you request a dump on exceptions, `MaxDumpCount` lets you configure the maximum number of dumped files, and `DumpPath` lets you specify where to save the dumped image files.

B.9 Application Partitioning

Application partitioning is an intricate new service aimed to refine and improve management of COM+ applications in a large-scale environment. An in-depth discussion of application partitions is beyond the scope of this appendix and requires an understanding of Active Directory. Instead, this appendix provides a simplified overview of the partition concept.

An *application partition* is a group of COM+ 1.5 applications. Partitions provide you with an economic way to present each user (be it a logged-on user or a call coming in across the network) with its own set of applications and components. Partitions are usually configured in Active Directory.

Under COM+ 1.0, a component can belong to only one COM+ application on a given machine. If you want to install the same component (same CLSID) in multiple applications, you have to do so on multiple machines. COM+ partitions allow you to install the same component in more than one application, provided the applications belong to different partitions. A given machine can contain multiple partitions, and each partition can have its own set of applications and components. You can assign users to partitions in the Active Directory. COM+ 1.5 also defines a *base partition*—a partition that all users share. When a user tries to create a component, COM+ first looks in the partition the user is associated with. If that partition has that component, then COM+ creates it. If it does not, COM+ looks in the base partition; if it is found in the base partition, COM+ creates it. If the base partition does not contain the component, then the creation fails, even if the component is part of another partition.

For example, consider the partition layout in Figure B-11. If a user associated with Partition A only tries to create the component with CLSID1, that component is created from Partition A; the configuration settings of App1 in Partition A and any component-level configuration are applied. However, if the user tries to create the component with CLSID3, the component from the base partition is created and the base partition settings are applied. If the user tries to create with CLSID7, the creation fails.

Figure B-11. Configuring multiple sets of applications on the same machine using partitions

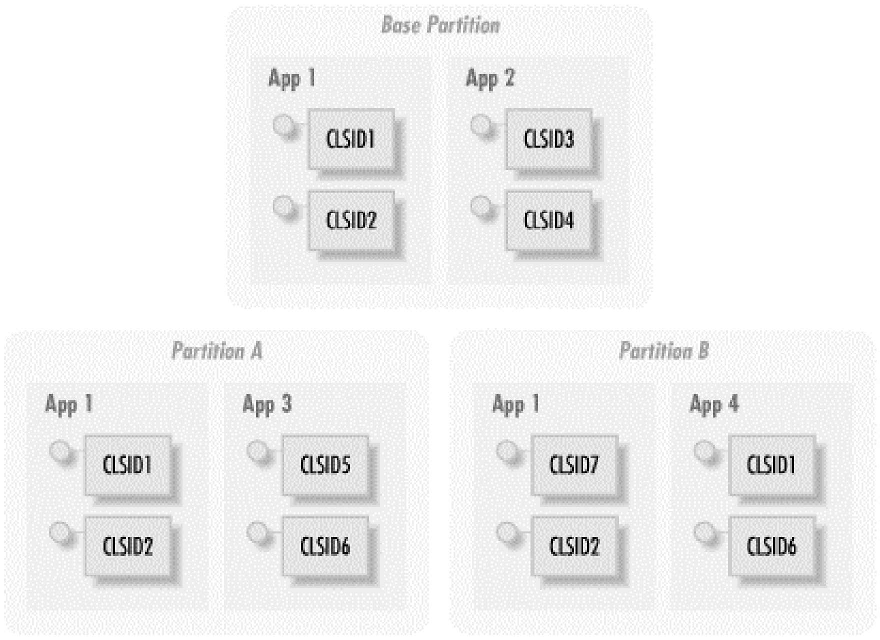


Figure B-11 demonstrates some other points. A given CLSID can belong to more than one partition, but a given partition can have only one copy of the component. Different partitions can contain applications with the same name. The different partitions inherit the base partition's applications and components, but they can override them, remove them, add new components, and change their settings.

Application partitions provide an easier way to manage activations and isolate applications between partitions. Each partition can be managed independently of the others, and you can even install different versions of the same component in different partitions, thus tailoring a particular compatibility solution.

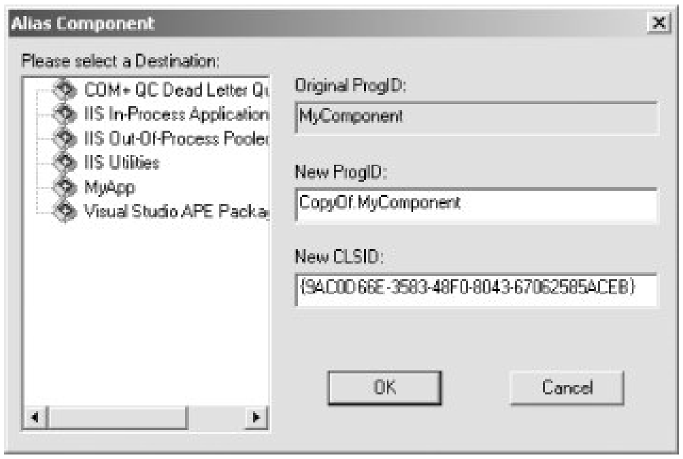
Under COM+ 1.5, the object context has a partition property. The context object supports a new interface called `IObjectContextInfo2` that derives from `IObjectContextInfo`, which enables you to get information about the partition and the application the object is part of. Clients can request to create an object in a particular partition using a special moniker.

The `ICOMAdminCatalog2` interface provides you with numerous methods for managing partitions, including copying an application from one partition to another, copying and moving a component from one partition to another, getting the base application partition ID, and getting the current partition ID.

B.10 Aliasing Components

Under COM+ 1.0 you cannot use the same component with more than one set of configurations—like in classic COM, a component is associated with just one CLSID. COM+ 1.5 allows you to alias an existing configured component with a new CLSID and apply a new set of configurations to the "new" component. This process is called *aliasing* a component. Aliasing is often a useful feature—you can develop a piece of business logic and assign more than one set of configuration parameters to it by copying it as many times as you like. The component's client can now decide which configuration setting and business logic implementation to instantiate by creating a particular CLSID. To alias a component, select Alias from its pop-up context menu in the Component Services Explorer. This selection brings up the Alias Component dialog box (see Figure B-12)

Figure B-12. Aliasing a component



Original ProgID: MyComponent

New ProgID: CopyOf.MyComponent

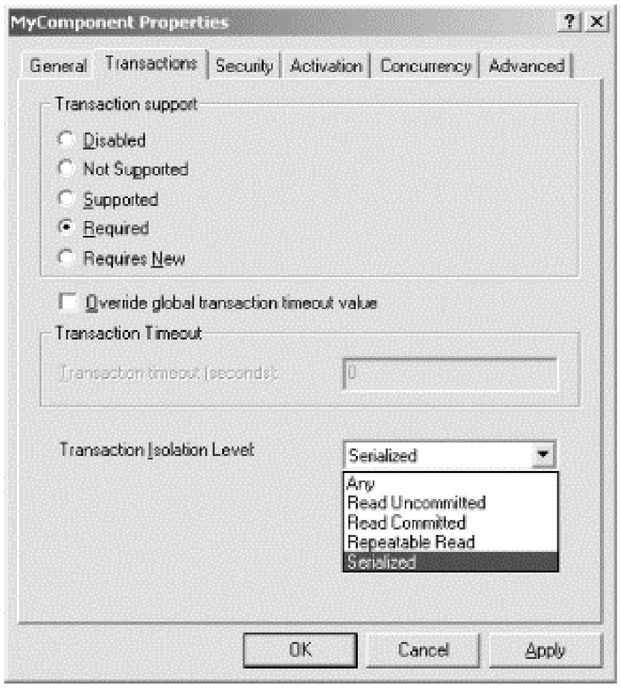
New CLSID: {9AC0D66E-3583-48F0-8043-67062585ACEB}

The dialog box lets you select a destination application for the new component. Because you are assigning a new CLSID to the component, you can even copy it back to its current application. The dialog generates the new CLSID for the copy and a new prog-ID (*<CopyOf>.<Old prog-ID>*, see Figure B-12). You can provide your own values for the CLSID and prog-ID, if you like. Initially, the new component has configuration settings that are identical to the original component. Once you copy a component, the original and the clone are considered different components from the COM+ point of view. You can configure them differently, even though the configurations apply to the same actual component at runtime. Copying components is also handy in the case of event classes. As you may recall from Chapter 9, you must supply COM+ with a skeletal implementation of an event class (stub out all implementation of the sinks) so that COM+ can synthesize its own implementation of the event class. You may often provide more than one event class so that some subscribers can subscribe to one event class and some to another. With component copying, you only need to provide one, and then just copy it.

B.11 Configurable Transaction Isolation Level

COM+ 1.0 handles transaction isolation very conservatively. COM+ 1.0 only allows the highest level of isolation, an isolation level called *serialized*. With serialized transactions, the results obtained from a set of concurrent transactions are identical to the results obtained by running each transaction serially. Such a high degree of isolation comes at the expense of overall system throughput; the resource managers involved have to hold onto both read and write locks for as long as a transaction is in progress, and all other transactions are blocked. However, you may want to trade system consistency for throughput in some situations by lowering the isolation level. Imagine, for example, a banking system. One of the requirements is to retrieve the total amount of money in all customer accounts combined. Although executing that transaction with the serialized isolation level is possible, if the bank has hundreds of thousands of accounts, it may take quite a while to complete. The transaction may time out and abort because some accounts may be accessed by other transactions at the same time. But the number of accounts could be a blessing in disguise. On average, statistically speaking, if the transaction is allowed to run at a lower isolation level, it may get the wrong balance on some accounts. However, those incorrect balances would tend to cancel each other out. The actual resulting error may be acceptable for the bank's need.
COM+ 1.5 allows you to configure the isolation level for a transactional component. The Transactions tab has a drop-down list box with five isolation levels (see Figure B-13). The available isolation levels are Any, Read Uncommitted, Read Committed, Repeatable Read, and Serialized. The default is set to Serialized.

Figure B-13. Setting transaction isolation level for individual components


MyComponent Properties
General | Transactions | Security | Activation | Concurrency | Advanced
OK | Cancel | Apply

Transaction support

Disabled

Not Supported

Supported

[✓ Required]

Requires New

Override global transaction timeout value

Transaction Timeout

Transaction timeout (seconds): 0

Transaction Isolation Level: Serialized

Any

Read Uncommitted

Read Committed

Repeatable Read

[✓ Serialized]

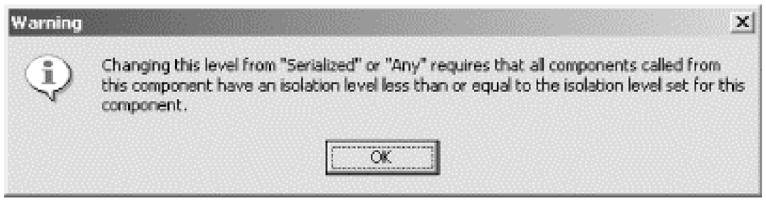
The underlying transaction processing monitor, the DTC, supports other transaction isolation levels besides Serialized, but COM+ 1.0 passes in a hard-coded isolation level of Serialized when it creates a new DTC transaction. All COM+ 1.5 does to expose these levels is pass the configured isolation level, instead of the original hard-coded Serialized level in COM+ 1.0, to the DTC.

Selecting an isolation level other than Serialized is commonly used for read-intensive systems. It requires a solid understanding of transaction processing theory and the semantics of the transaction itself, the concurrency issues involved, and the consequences for system consistency. A good starting point is the bible on transaction processing: *Transaction Processing: Concepts and Technologies* by Jim Gray and Andreas Reuter (Morgan Kaufmann, 1992). In addition, not all resource managers support all levels of isolation, and they may elect to take part in the transaction at a higher level than the one configured. Every isolation level besides Serialized is susceptible to some sort of inconsistency resulting from having other transactions access the same information. The difference between the four isolation levels is in the way they use read and write locks. A lock can be held only when the transaction accesses the data in the resource manager, or it can be held until the transaction is committed or aborted. The former is better for throughput; the latter for consistency. The two kinds of locks and the two kinds of operations (read/write) give four isolation levels. See a transaction-processing textbook for a comprehensive description of isolation levels.

In a COM+ transaction, the root does more than just start and end a transaction. It also determines the isolation level for the transaction. Once determined, the isolation level is fixed for the life of the transaction. A component cannot take part in a transaction if the isolation level it requires is greater than that of the transaction. Consequently, every component in a transaction must have its isolation level set to a level equal to or less than that of the root component. If a component in a transaction tries to create another component with a greater isolation level, COM+ 1.5 refuses to create the component and `CoCreateInstance( )` returns `CO_E_ISOLEVELMISMATCH`.

When isolation is set to Any, the component is indifferent to the isolation level of the transaction it is part of. If that component is not the root of a transaction, then it simply assumes the isolation level of the transaction it is part of when it accesses resource managers. If that component is the root of a transaction, then COM+ 1.5 decides on the isolation level for it and uses Serialized. As a result, any component with isolation set to Serialized or Any can be the root of a COM+ 1.5 transaction because by definition, all other components have isolation levels equal to or less than they do. Any other isolation level for a root may not guarantee successful activation of internal components. The COM+ 1.5 Explorer displays a warning message when you change isolation level from Serialized or Any, which is almost correct in its content (see Figure B-14).

Figure B-14. Warning message when changing the isolation level from Serialized or Any to another level



Changing this level from "Serialized" or "Any" requires that all components called from this component have an isolation level less than or equal to the isolation level set for this component.

It is possible for one component to call another component with a higher configured isolation level, as long as the transaction isolation is greater than or equal to that higher level. For example, Component R with isolation set to Repeatable Read is the root of a transaction, and it creates two other components, A and B, with isolation levels of Read Committed and Read Uncommitted, respectively. Component B can call Component A because the isolation level of A and B is less than that of the root R.

The correct warning message should read: "Changing this level from Serialized or Any requires that when this component is the root of a transaction, all components in the transaction have an isolation level less than or equal to the isolation level set for this component."

You can also set the component's isolation level programmatically by setting the `TxIsolationLevel` named property of a component catalog object.

NET Serviced Component Isolation

A .NET transactional serviced component can declare its isolation level under COM+ 1.5 using the `Transaction` attribute's `Isolation` property:

```
[Transaction(Isolation=
TransactionIsolationLevel.Serializable)]
public class MyComponent :ServicedComponent
{}
```

The `Isolation` property is of the enum type `TransactionIsolationLevel`, defined as:

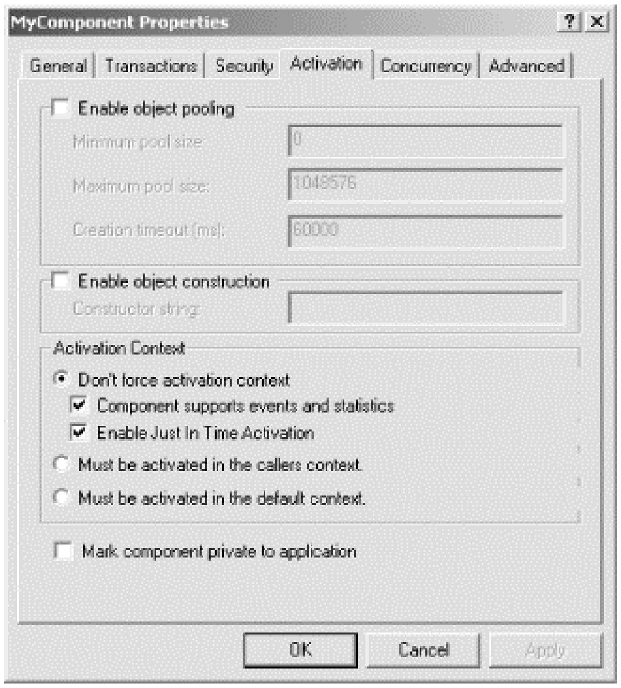
```
public enum TransactionIsolationLevel
{
    Any,
    ReadUncommitted,
    ReadCommitted,
    RepeatableRead,
    Serializable
}
```

The default value of the `TransactionIsolationLevel` property is `TransactionIsolationLevel.Serializable`.

B.12 Improved Context Activation Setting

As explained in Chapter 3, configuring your component to use JITA requires having its own context. COM+ 1.0 lets you configure your component to use JITA, and configure it to require that the component always must be activated in its creator's context, by checking the checkbox "Must be activated in caller's context" on the component's Activation tab. (As discussed in Chapter 3, this name is inaccurate and should read "Must be activated in creator's context.") These two settings are mutually exclusive. If you configure a component in this way, all activations fail. You face a similar pitfall when configuring the component to use transactions or enforce security access checks for the component—all require a private context. The COM+ 1.5 Explorer remedies this situation by redesigning the component Activation tab (see Figure B-15) and adding a new activation option. The Activation Context properties group contains three radio buttons. You can select only one of the buttons at a time—thus enforcing mutual exclusion. If you select "Don't force activation context," you actually select the regular COM+ context activation behavior. In this mode, you can enable JITA, transactions, and security access checks. In fact, as long as transaction support or access security are enabled, you cannot select another option; enabling security checks sets the selection back to "Don't force activation context" from any other setting. COM+ 1.5 adds a new context activation selection—"Must be activated in the default context." This new option can be useful when you know that clients of the component reside in the default context and make frequent calls of short duration to your component, and that your component does not use most of the COM+ services.

Figure B-15. The new COM+ 1.5 component Activation tab



☐ Enable object pooling
Minimum pool size: 0
Maximum pool size: 1048576
Creation timeout (ms): 60000

☐ Enable object construction
Constructor string:

Activation Context
● Don't force activation context
☑ Component supports events and statistics
☑ Enable Just In Time Activation
○ Must be activated in the callers context.
○ Must be activated in the default context.

☐ Mark component private to application

B.13 Private Components

COM+ 1.5 provides a new feature called *private components*. Every component has, at the bottom of its activation tab, the "Mark component private to application" checkbox (see Figure B-15). A private component can only be accessed by other components in the same application. Private components are needed in almost every decent size COM+ project. To promote loose coupling between clients and objects, you should avoid providing the clients with access to the internal objects by marking them as private.

.NET Private Serviced Component

A .NET transactional serviced component can declare itself as a private component, using the PrivateComponent attribute:

```
[PrivateComponent]
public class MyComponent :ServicedComponent
{}
```

Note that a private component is different from an internal component. Declaring the class as internal instead of public prevents access to it from outside its assembly. A private component cannot be accessed by clients outside its COM+ application, but it can be accessed by other clients in the same application, including components from other assemblies.

B.14 Web Services in COM+ 1.5

Web services support is the most exciting new feature of the .NET platform. As explained in Chapter 10, *web services* allow a middle-tier component in one web site to invoke methods on another middle-tier component at another web site, with the same ease as if the two components were on the same site and machine. But .NET web services come with a price—companies have to invest in learning new languages such as C# and cope with a new programming model and new class libraries. In most organizations, this cost is substantial. To preserve existing investment in COM components and development expertise, while providing a migration path to the .NET world, COM+ 1.5 can expose any COM+ component that complies with web services design guidelines as a web service. The application activation tab lets you configure SOAP activation for your application (see Figure B-7). All you need to do is specify the virtual directory of the web service associated with this application, and COM+ provides the necessary adaptors as a component service. Each component is exposed as a separate web service, identified by the component prog-ID under the virtual directory. COM+ installs the web services with IIS and generates the proper web service configuration and information files. Note that IIS and .NET must be installed on the server and client machine to enable the SOAP activation mode for your application.

B.15 Summary

COM+ is essential for rapid component development and robust, scalable applications. COM+ 1.5 smoothes COM+ 1.0's rough edges, and its new features are a most welcome addition to your development arsenal. Future releases of COM+ will most likely introduce new features and component services, probably to complement new capabilities available with .NET (such as the web services support) and improve the integration between the two. Especially noteworthy is COM+ 1.5's support for legacy components and applications. The message is clear: use COM+ as a supporting component services platform and unify in the same architecture all your components—from classic COM components, to COM+ configured components, to .NET serviced components.

As mentioned at the beginning of the book, COM+ offers a migration path for companies and developers. Companies can start (or continue) their projects in COM, using COM+ for component services. When the time comes to move to .NET, they can start plugging into the same architecture .NET serviced components in a seamless manner, reusing and interacting with their existing COM and COM+ configured components.

Appendix C. Introduction to .NET

.NET is based on a *Common Language Runtime* (CLR) environment that manages every runtime aspect of your code. All .NET components, regardless of the language in which they are developed, execute in the same runtime (hence the name). The CLR is like a warm blanket that surrounds your code, providing it with memory management, a secure environment to run in, object location transparency, concurrency management, and access to the underlying operating system services. Because the CLR manages these aspects of your object's behavior, code that targets the CLR is called *managed code.* The CLR provides absolute language interoperability, allowing a high degree of component interoperability. COM, too, provides language independence, allowing binary components developed in two different languages (such as Visual Basic and C++) to call one another's methods, but COM language interoperability is only at runtime. During development, .NET allows a component developed in one language to derive from a component developed in another language seamlessly. .NET is capable of this process because the CLR is based on a strict type system. To qualify as a .NET language, all constructs (such as class, struct, or primitive types) in every language must compile to CLR-compatible types. The language interoperability gain is at the expense of existing languages and compilers. Existing compilers produce CLR-ignorant code—code that does not comply with the CLR type system and that is not managed by the CLR. Visual Studio.NET comes with four CLR-compliant languages: C#, Visual Basic.NET, JScript.NET, and Managed C++. Third-party compiler vendors also target the CLR, with more than 20 other languages, from COBOL to Eiffel.

C.1 .NET Programming Languages

All .NET programming languages use the same set of base classes, development environment, and CLR types and comply with the same CLR design constraints. Compiling code in .NET is a two-phase process. First, the high-level code is compiled into a generic machine-code-like language called *intermediate language* (IL). At runtime, on the first call into the IL code, the IL is compiled into native code and executes as native code. The native code is used until the program terminates. The IL is the common denominator of all .NET programming languages, and equivalent constructs in two different languages should theoretically produce identical IL. As a

result, all .NET programming languages are equal in performance and ease of development.
The difference between the languages is mostly aesthetic, and choosing one over another is a matter of personal preference. For example, to make C++ CLR compliant, Microsoft had to add numerous nonstandard compiler directives and extensions, resulting in less readable code than standard unmanaged C++. Similarly, Visual Basic.NET bears little resemblance to its Visual Basic 6.0 ancestor, requiring you to unlearn things you used to do in Visual Basic 6.0. Only C# has no legacy and is a fresh .NET language. C# is a C++ derivative language, combining the power of C++ with the ease of Visual Basic 6.0, and offering you readable, CLR-compliant C++ like code. In fact, C# looks more like normal C++ than managed C++. This appendix and Chapter 10 therefore use C# in its code samples. Bear in mind, however, that you can do all the code samples in Visual Basic.NET, managed C++, or any other .NET language.
Other features of .NET languages include their treatment of every entity as an object (including primitive types), resulting in a cleaner programming model. .NET provides common error handling based on exceptions. The CLR has a rich predefined set of exception classes that you can use as is, or derive and extend for a specific need. An exception thrown in one language can be caught and handled in another language.

C.2 Packaging .NET Components: Assemblies

The basic code packaging unit in .NET is the *assembly*. An assembly is a *logical* DLL—i.e., assembly can combine more than one physical DLL into a single deployment, versioning, and security unit. However, an assembly usually contains just one DLL (the default in Visual Studio.NET) and you have to use command-line compiler switches to incorporate more than one DLL in your assembly. An assembly is not limited to containing only DLLs. An assembly can also contain an EXE. As a component developer, you usually develop components that reside in a single or multiple DLL assembly to be consumed by a client application residing in an assembly that has an EXE. The code in the assembly (in the DLLs or the EXE) is only the IL code, and at runtime the IL is compiled to native code, as explained previously.
An assembly contains more than just the IL code. Embedded in every assembly is *metadata,* a description of all the types declared in the assembly and a *manifest,* a description of the assembly and all other required assemblies. The manifest contains various assembly-wide information, such as the assembly version

information. The version information is the product of a version number provided by the developer and a build and revision number captured by the compiler (or provided by the developer as well) during the build. All DLLs in the assembly share the same version number and are deployed as one unit.
The assembly boundary serves as the .NET security boundary—security permissions are granted at the assembly level. All components in an assembly share the same set of permissions.
The assembly can also contain a compiled resource file for icons, pictures, etc., like any traditional DLL or EXE.

C.3 Developing .NET Components

To create a .NET component in C# (or any other .NET Language), you simply declare a class. When the class is instantiated by the CLR, the result is a binary component. Example C-1 shows a simple class named `MyClass` that implements the `IMessage` interface and displays a message box with the word "Hello" when the interface's `ShowMessage( )` method is called.

Example C-1. Building a component in .NET

```
namespace MyNamespace
{
   using System;
   using System.Windows.Forms;

   public interface IMessage
   {
      void ShowMessage(  );
   }

   public class MyComponent :IMessage
   {
      public MyComponent(){}//constructor
      ~ MyComponent(){}//destructor
      public void ShowMessage(  )
      {
         MessageBox.Show("Hello!","MyComponent");
      }
   }
 }
```

The `MyComponent` class in Example C-1 is defined as `public`, making it accessible to any .NET or COM client once you export the component to COM. You can define a class constructor to do object initialization, as in this example, but the destructor has different semantics than the classic C++ destructor because .NET uses

nondeterministic object finalization. You can implement other methods to do object cleanup as well. The implementation of `ShowMessage( )` uses the static `Show( )` method of the `MessageBox` class. Like in C++, C# allows you to call a class (static) method without instantiating an object first.
Example C-1 demonstrates a few additional key points regarding developing .NET components: using namespaces and interface-based programming. These points are discussed next.

C.3.1 Using Namespaces

The class definition is scoped in a *namespace.* Namespaces are optional, but you are encouraged to use them. Namespaces in .NET have the same purpose they have in C++: to scope classes so a client can use different classes from different vendors that have the same name. For a namespace, you typically use the product's name, your company's name, or the assembly's name. A client of the class `MyComponent` in Example C-1 must now refer to it by qualifying it with its containing namespace:

```
MyNamespace.MyComponent
```

Alternatively, the client can say that it is using the `MyNamespace` namespace, and avoid putting the "MyNamespace" prefix on every type contained in that namespace:

```
using MyNamespace;
//MyComponent is now the same as MyNamespace.MyComponent
```

You can also nest namespaces within one another. For example, if your company develops more than one product, you would typically define in the scope of the `MyCompany` namespace, the nested namespaces `Product1`, `Product2`, and so on:

```
namespace MyCompany
{
   namespace Product1
   {
      //classes and type definitions
      public class Component1
      {...}
   }
   namespace Product2
   {
      //other classes and type definitions
   }
}
```

Clients of your components must give the full qualifying namespace to access your component:

```
MyCompany.Product1.Component1
```

Or, clients can use the `using` statement:

```
using MyCompany.Product1;
```

```
//Component1 is now the same as
MyCompany.Product1.Component1
```
The `ShowMessage( )` method in Example C-1 uses the static method `Show( )` of the `MessageBox` class, defined in the `System.Windows.Forms` namespace. Example C-1 therefore contains the statement:
```
using System.Windows.Forms;
```
This statement is used to simplify downstream code.

C.3.2 Using Interfaces

One the most important principles of component-oriented development is the separation of interfaces from implementation. COM enforces this separation by having you separate the definitions of interfaces and classes. .NET does not force you to have your class methods be part of any interface, but it is imperative that you do so to allow polymorphism between different implementations of the same interface.
Example C-1 includes an interface definition as part of the code—there is no need for a separate IDL file. The reserved C# word `interface` allows you to define a type that is purely virtual (it has no implementation and cannot be instantiated by clients), just like a C++ pure virtual or abstract class. The interface methods do not have to return `HRESULT` or any other error handling type. In case of an error, the method implementation should throw an exception.

C.4 Writing .NET Client-Side Code

All that a .NET client has to do to use a component is add a reference in its project setting to the assembly containing the component, create an object, and then use it:
```
using MyNamespace;

//Interface-based programming:
IMessage myObj;
myObj = (IMessage)new MyComponent(  );
myObj.ShowMessage(  );
```
You usually do not use pointers in C#. Everything is referenced using the dot (.) operator. Note also that the client casts the newly created object to the `IMessage` interface. This is the .NET equivalent of `QueryInterface( )`. If the object does not support the interface it is being cast to, an exception is thrown.
The client can instead perform a controlled query for the interface using the `as` keyword. If the object does not support the interface, the returned reference is `null`:
```
using MyNamespace;
```

```
//Even better: check for type mismatch
IMessage myObj;
myObj = new MyComponent(  ) as IMessage;
Debug.Assert(myObj!= null);
myObj.ShowMessage(  );
```

As mentioned before, .NET does not enforce separation of interface from implementation, so the client could create the object type directly:

```
using MyNamespace;

//Avoid doing this:
MyComponent myObj;
myObj = new MyComponent(  );
myObj.ShowMessage(  );
```

However, you should avoid writing client code that way because doing so means that the client code is not polymorphic with other implementations of the same interface. Such client code also couples interacting modules. Imagine a situation in which Module 1 creates the object and Module 2 uses it. If all that the Module 1 passes to Module 2 is the interface type, Module 1 can change the implementation of the interface later without affecting Module 2.

C.5 .NET as a Component Technology

To simplify component development, one of the goals set for the .NET framework was to improve COM deficiencies. Some of these deficiencies, such as awkward concurrency management via apartments, were inherited with COM itself. Other deficiencies occur as a result of error-prone development and deployment phases. Examples include memory and resource leaks resulting from reference count defects, fragile registration, the need for developer-provided proxy stubs pairs, and having interface and type definition in IDL files separate from the code. Frameworks such as ATL do provide automation of some of the required implementation plumbing, such as class factories and registration, but they introduce their own complexity.

.NET is designed to not only improve these deficiencies, but also maintain the core COM concepts that have proven themselves as core principles of component-oriented development.

.NET provides you fundamental component-oriented development principles, such as binary compatibility between client and component, separation of interface from implementation, object location transparency, concurrency management, security, and language independence. A comprehensive discussion of .NET as a component technology merits a book in its own right and is beyond the scope of this appendix. However, the following sections describe the main characteristics of .NET as a component technology.

C.5.1 Simplified Component Development

Compared to COM, .NET might seem to be missing many things you take for granted as part of developing components. However, in essence, the missing elements are actually present in .NET, although in a different fashion:

- There is no canonical base interface (such as `IUnknown`) that all components derive from. Instead, all components derive from the `System.Object` class. Every .NET object is therefore polymorphic with `System.Object`.
- There are no class factories. In .NET, the runtime resolves a type declaration to the assembly containing it and the exact class or struct within the assembly.
- There is no reference counting of objects. .NET has a sophisticated garbage collection mechanism that detects when an object is no longer used by clients. Then the garbage collector destroys the object.
- There are no IDL files or type libraries describing your interfaces and custom types. Instead, you put those definitions in your source code. The compiler is responsible for embedding the type definitions in a special format in your assembly called metadata.
- There are no GUIDs. Scoping the types with the namespace and assembly name provides uniqueness of type (class or interface). When sharing an assembly between clients, the assembly must contain a *strong name*—a unique binary blob generated with an encryption key. Globally unique identifiers do exist in essence, but you do not have to manage them anymore.
- There are no apartments. By default, every .NET component executes in a free-threaded environment and you are responsible for synchronizing access to your components. Providing synchronization is done by either relying on .NET synchronization locks or using COM+ activities.

.NET has a superb development environment and semantics, the product of years of observing how developers use COM and the hurdles they faced.

C.5.1.1 The .NET base classes

As demonstrated in Example C-1, a hard-to-learn component development framework such as ATL is not required to build binary managed components. .NET takes care of all the underlying plumbing for you. To help you develop your business logic faster, .NET also provides you with more than 3,500 base classes, available in similar form for all languages. The base classes are easy to learn and apply. You can use the base classes as is, or derive from them to extend and specialize their behavior.

C.5.1.2 Component inheritance

.NET enforces strict inheritance semantics and inheritance conflicts resolution. .NET does not allow multiple inheritance of implementation. You can only derive from one concrete class. You can, however, derive from as many interfaces as you like. When you override a virtual function implementation in a base class, you must declare your intent explicitly. For example, if you want to override it, you should use the `override` reserved word.

C.5.1.3 Component visibility

While developing a set of interoperating components, you often have components that are intended only for private use and should not be shared with your clients. Under COM, there is no easy way of guaranteeing that the components are only used privately. The client can always hunt through the Registry, find the CLSID of your private component, and use it. In .NET, you can simply use the `internal` keyword on the class definition (instead of `public`, as in Example C-1). The runtime denies access to your component for any caller outside your assembly.

C.5.1.4 Attribute-based programming

When developing components, you can use attributes to declare your component needs, instead of coding them. Using attributes to declare component needs is similar to the way COM developers declare the threading model attribute of their components. .NET provides you with numerous attributes, allowing you to focus on your domain problem at hand (COM+ services are accessed via attributes). You can also define your own attributes or extend existing ones.

C.5.1.5 Component-oriented security

The classic Windows NT security model is based on what a given user is allowed to do. This model has evolved in a time when COM was in its infancy and applications were usually standalone, monolithic chunks of code. In today's highly distributed, component-oriented environment, there is a need for a security model based on what a given piece of code—a component—is allowed to do, and not only on what its caller is allowed to do.

.NET allows you to configure permissions for a piece of code and provide an *evidence* to prove that it has the right credentials to access a resource or perform sensitive work. Evidence-based security is tightly related to the component's origin. System administrators can decide that they trust all code that came from a particular vendor, but distrust everything else, from downloaded components to malicious attacks. A component can also demand that a permission check be performed to verify that all callers in its call chain have the right permissions before it proceeds to do its work.
This model complements COM+ role-based security and call authentication. It provides the application administrator with granular control over not only what the users are allowed to do, but also what the components are allowed to do. .NET has its own role-based security model, but it is not as granular or user friendly as COM+ role-based security.

C.5.2 Simplified Component Deployment

.NET does not rely on the Registry for anything that has to do with your components. In fact, installing a .NET component is as simple as copying it to the directory of the application using it. .NET maintains tight version control, enabling *side-by-side* execution of new and old versions of the same component on the same machine. The net result is zero-impact install—by default, you cannot harm another application by installing yours, thus ending the predicament known as *DLL Hell*. The .NET motto is: it just works. If you want to install components to be shared by multiple applications, you can install them in the *Global Assembly Cache* (GAC). If the GAC already contains a previous version of your assembly, it keeps it for use by clients that were built against the old version. You can purge old versions as well, but that is not the default.

C.5.3 Simplified Object Life Cycle Management

.NET does not use reference counting to manage an object's life cycle. Instead, .NET keeps track of accessible paths in your code to the object. As long as any client has a reference to an object, it is considered *reachable*. Reachable objects are kept alive. Unreachable objects are considered garbage, and therefore destroying them harms no one. One of the crucial CLR entities is the *garbage collector*. The garbage collector periodically traverses the list of existing objects. Using a sophisticated pointing schema, it detects unreachable objects and releases the memory allocated to these objects. Consequently, clients do not have to increment or decrement a reference count on the objects they create.

C.5.4 Nondeterministic Finalization

In COM, the object knows that it is no longer required by its clients when its reference count goes down to zero. The object then performs cleanup and destroys itself by calling `delete this;`. The ATL framework even calls a method on your object called `FinalRelease( )`, letting you handle the object cleanup.
In .NET, unlike COM, the object itself is never told when it is deemed as garbage. If the object has specific cleanup to do, it should implement a method called `Finalize( )`. The garbage collector calls `Finalize( )` just before destroying the object. `Finalize( )` is your .NET component's destructor. In fact, even if you implement a destructor (such as the one in Example C-1), the compiler will convert it to a `Finalize()` method.

C# Destructor

In C#, do not provide a `Finalize( )` method. Instead, provide a destructor. The compiler both converts the destructor definition to a `Finalize( )` method and calls your base class `Finalize( )` method.
For example, for this class definition:

```
public class MyClass
{
    public MyClass(  ){}
    ~MyClass(  ){}
}
```

The code that is actually generated would be:

```
public class MyClass
{
    public MyClass(  ){}
    protected virtual void Finalize(  )
    {
         try
       {
          //Your destructor code goes here
       }
       finally
       {
          base.Finalize(  );//everybody has one,
from Object
       }
    }
}
```

However, simplifying the object lifecycle comes with a cost in system scalability and throughput. If the object holds on to expensive resources, such as files or database connections, those resources are released only when `Finalize( )` is called. It is called at an undetermined point in the future, usually when the process hosting your component is out of memory. In theory, releasing the

expensive resources the object holds may never happen, and thus severely hamper system scalability and throughput.
There are two solutions to the problems arising from nondeterministic finalization. The first solution is to implement methods on your object that allow the client to explicitly order cleanup of expensive resources the object holds. If the object holds onto resources that can be reallocated, then the object should expose methods such as `Open( )` and `Close( )`.
An object encapsulating a file is a good example. The client calls `Close( )` on the object, allowing the object to release the file. If the client wants to access the file again, it calls `Open( )` without re-creating the object. The more common case is when disposing of the resources amounts to destroying the object. In that case, the object should implement a method called `Dispose( )`. When a client calls `Dispose( )`, the object should dispose of all its expensive recourses, and the client should not try to access the object again. The problem with both `Close( )` and `Dispose( )` is that they make sharing the object between clients much more complicated than COM's reference counts. The clients have to coordinate which one of them is responsible for calling `Close( )` or `Dispose( )` and when `Dispose( )` should be called; thus, the clients are coupled to one another.
The second solution to nondeterministic finalization is to use COM+ JITA, as explained in Chapter 10.

C.5.5 COM and Windows Interoperability

COM and .NET are fully interoperable. Any COM client can call your managed objects, and any COM object is accessible to a managed client. To export your .NET components to COM, use the *TlbExp.exe* utility, also available as a command from the Tools menu. The utility generates a type library that COM clients use to CoCreate managed types and interfaces. You can use various attributes on your managed class to direct the export process, such as providing a CLSID and IID.
To import an existing COM object to .NET (by far the most common scenario), use the *TlbImp.exe* utility. The utility generates a managed wrapper class, which your managed client uses. The wrapper manages the reference count on the actual COM object. When the wrapper class is garbage collected, the wrapper releases the COM object it wraps. You can also import a COM object from within the Visual Studio.NET environment by selecting the COM object from the project reference dialog (which makes Visual Studio.NET call *TlbImp* for you).
.NET has support for invoking native Win32 API calls, or any DLL exported functions, by importing the method signatures to the managed environment.

C.6 Composing Assemblies

You provide the compiler with your assembly information in an assembly information file (usually called in a C# project, *AssemblyInfo.cs*). The assembly information file is compiled with the rest of the project's source files. The information in the file is in the form of *assembly attributes*—directives to the compiler on the information to embed in the assembly. Example C-2 shows a typical set of assembly attributes.

Example C-2. The assembly information file includes a variety of assembly attributes

```
[assembly: AssemblyTitle("MyAssembly")]
[assembly: AssemblyDescription("Assembly containing demo
.NET components")]
[assembly: AssemblyCompany("My Product")]
[assembly: AssemblyCopyright("(c) 2001 My Company ")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("en-US")]
[assembly: AssemblyVersion("1.0.*")]
```

C.6.1 Sharing Assemblies

Assemblies can be private or shared. A *private* assembly resides in the same directory of the application that uses it (or in its path). A *shared* assembly is in a known location, called the *global assembly cache* (GAC), mentioned in Chapter 10Chapter 10. To add an assembly to the GAC, use either the .NET administration tool or the *GACUtil* command-line utility. Once in the GAC, the assembly can be accessed by multiple applications, both managed and unmanaged. To avoid conflicts in the GAC between different assemblies that have the same name, a shared assembly must have a *strong name*. The strong name authenticates the assembly's origin and identity and cannot be generated by a party other than the original publisher. The strong name allows any client of the assembly (usually the assembly loader) to deterministically verify that the assembly was not tampered with. Assigning a strong name to an assembly is also known as *signing* the assembly. To assign a strong name to your assembly, you first need to generate private or public encryption keys. You can generate the pair using the *SN.exe* command-line utility:

```
SN.exe -k MyAssembly.snk
```

Future versions of Visual Studio.NET may enable you to generate keys from within the visual environment. The *-k* switch instructs SN to generate a new pair of keys and store them in the filename

specified. The convention used for the filename is the assembly name with the strong name key (*snk)* extension, but it can actually be any name and extension you like.
You then add the *snk* file to the assembly's information file, using the `AssemblyKeyFile` assembly attribute:

```
[assembly:AssemblyKeyFile("MyAssembly.snk")]
```

In addition to a version number and a strong name, a shared assembly must have a namespace and *locale* identifier that identify the human language used in its user interface. In Example C-2 the locale is specified by the `AssemblyCulture` assembly attribute.

C.6.2 Assembly Metadata

Each assembly must contain metadata. The metadata is the .NET equivalent of COM's type libraries, except the metadata is more like a type library on steroids. The metadata contains descriptions of all the types defined in the assembly, such as interfaces, classes and their base classes, method signatures, properties, events, member variables, and custom attributes. The metadata is generated automatically by the compiler when it compiles the source files of your project. You can view the metadata of your assembly using the *ILDASM* utility.

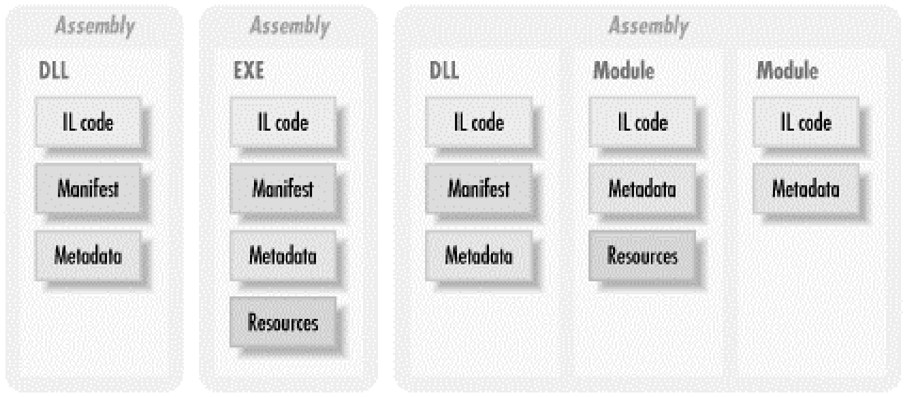
C.6.3 Assembly Manifest

While the metadata describes the types in your assembly, the manifest describes the assembly itself. The manifest contains the assembly version information, the locale information, and the assembly's strong name. The manifest also contains the visibility of the assembly's types—which types are public (can be accessed by other assemblies) and which types are internal (can only be accessed from within the assembly). Finally, the manifest contains the security permission checks to run on behalf of the assembly. Like the metadata, the manifest is generated automatically by the compiler during assembly compilation. You can view the manifest of your assembly using the *ILDASM* utility.

C.6.4 Assembly Files

Because every assembly must contain the manifest and metadata (and usually IL code and resources), a single DLL or EXE assembly contains all of them in one file. However, the only requirement of a multifile assembly is that a file containing IL must also contain metadata describing it. Such a file is called a *module.* A multifile assembly must still have one DLL file that contains the manifest. Figure C-1 shows a few possibilities for composing assemblies.

Figure C-1. Assembly files



As you can see, you can compose the assembly in almost any way and use compiler switches to bind all your files together. In practice, most assemblies contain just one DLL (the Visual Studio.NET IDE provides only this option) and are composed of one file.

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.
The animals on the cover of *COM and .NET Component Services* are moray and conger eels. Eels make up the 10 families of fish belonging to the order *Arguilliformes*. Known for their snakelike body with no hind fins, eels can move through water, mud, and rocky crevices. Most eels are less than three feet long, but freshwater conger eels can grow as large as nine feet. Until the 20th century, little was known about the life cycle and migration of eels. Scientists now know that American and European eels travel long distances during their reproductive cycles. The female eels generally mature in freshwater lakes and travel to the nearest ocean, often slithering over wet grass and mud during the journey. Then they swim or drift from Europe or North America to the Sargasso Sea. There, the females lay up to 20 million eggs and then die. The egg-larvae then drift either to North America (after one year) or back to Europe (after three years). After reaching their home continent, the eels complete their cycle by gathering at the mouths of rivers and swimming upstream. Eels are also known for their oily meat, cherished by some as a culinary delicacy.
Ann Schirmer was the production editor for *COM and .NET Component Services*. Paulette Miley and Ann Schirmer were the copyeditors for the book . Ann Schirmer and Leanne Soylemez were the proofreaders. Claire Cloutier, Mary Brady, and Rachel Wheeler provided quality control. Kimo Carter, Ann Schirmer, and Sarah Sherman did interior composition. Judy Hoer wrote the index.
Ellie Volckhausen designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Dover Pictorial Archive. Emma Colby produced the cover layout with Quark™ XPress 4.1 using Adobe's ITC Garamond font.
David Futato designed the interior layout. Neil Walls converted the files from Microsoft Word to FrameMaker 5.5.6 using tools created by Mike Sierra. The text font is Linotype Birka, the heading font is Adobe Myriad Condensed, and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Ann Schirmer.