T W E L V E

# An Introduction to the COM+ Services

With an understanding of the COM+ architecture under your belt, you are now ready to explore in greater depth each of the services that COM+ offers. COM+ provides fine-grained security, distributed transactions, enhanced scalability through object and connection pooling, thread synchronization, store and forward method invocation, and load balancing.

There are three steps you must perform to use these services in your COM components:

1. Create a COM+ application.
2. Add your COM components to the COM+ application.
3. Configure the components to use the COM+ services.

Steps 1 and 2 were covered in the previous chapter. Step 3 is the subject of this chapter. But which services should you use in a particular component? And what settings should you select for each of the attributes associated with a particular service? In this chapter, I go through the COM+ services one-by-one and explain how to configure them to implement your desired functionality.

## Fine-Grained Security

In Chapter 10 you learned about DCOM and how to configure security settings using a tool called `dcomcnfg`. `Dcomcnfg` is simple to use because it lets you configure your security settings declaratively (without writing code). The main problem with `dcomcnfg` is that it only lets you configure security settings for an entire server. All the COM classes in that server share the same security settings. This is not sufficient for servers that contain COM classes or interfaces that have very different security requirements. For instance, imagine that you had a COM class with two interfaces. One of these interfaces lets users submit Field Service (heating repair, air conditioning repair, and so on) work orders to an accounting system for billing. The other interface lets users view work orders. Only certain users (supervisors, billing clerks) should be allowed to submit work orders for billing, but anyone should be allowed to view work order information. To provide a solution for this scenario, you need to configure your security settings on a per-interface basis. Unfortunately, `dcomcnfg` does not provide this capability. In the past, the only way to implement fine-grained security in DCOM components was to use the COM security API. But this API is difficult to use and you cannot use it from 4GLs such as Visual Basic.

With COM+, you can control access to a server on a per-class, per-interface, or even a per-method basis. The best part is that COM+ performs this using attribute-based programming so you don't have to write any code. There are still some situations where using attribute-based programming alone is not sufficient. As an example, imagine you are developing software for a bank. At this fictional bank, a bank teller is allowed to make withdrawals from a customer's account as long as the transaction involves less than $1,000.00, but the approval of a bank manager is required for a withdrawal where the amount is greater than $1,000.00. You have to write code to enforce this kind of access control even if you are using COM+. You cannot implement this type of access control using attribute settings alone. Fortunately, COM+ provides a simple solution that lets you implement this kind of programmatic security logic easily and without having to resort to the DCOM security API. Whether you use COM+'s fine-grained security declaratively or programmatically, to understand COM+'s security features, you must first understand *roles*.

### *Roles*

COM+ implements a new security model that utilizes the concept of a *role*. A role is a group of users of a COM+ application that have the same security profile (they are allowed to perform the same operations and disallowed from performing the same operations). In most cases, the roles you define are the same as the real-life roles of the users who will use your applica-

tion. For instance, a COM+ application that is designed for a bank will most likely have a Teller role, a Manager role, and a Loan Office role. Each of these types of users share a security profile: they are allowed to do the same kinds of things and are disallowed from doing the same kinds of things. As a COM+ programmer, you define the roles that your application requires. The System Administrator responsible for your application decides which users will belong to each role after your application is deployed.

Once you have defined the roles for your application, you (or an administrator) can add users and groups to those roles (you'll see how to do this shortly). You can then use these roles two ways: using the Component Services Explorer, you can declaratively specify that only users in a particular role are allowed to use a particular component, interface, or method. You can also programmatically query your call context to see what role the current caller belongs to and throw an error if the user is trying to do something he is not allowed to do.

**CREATE ROLES**

You can add roles to your COM+ Application immediately after you create it. The financial component that you built in Chapter 7 has two interfaces: `ITimeValue` and `ITaxCalculation`.


**For this discussion, you can use the Financial component that you built in Chapter 7 and the TestApp COM+ Application you built in Chapter 11.**

Let's say that only Loan Officers should be allowed to use the methods in `ITimeValue` and only Tax Preparers should be allowed to use the methods in `ITaxCalculation`. To create these two roles, perform the following steps in the COM+ Component Services Explorer:

1. Right-click on the `Roles` Folder in your TestApp COM+ Application as shown in Figure 12.1.
2. Select `New\Role` from the `Context` menu (the New Role dialog is displayed as shown in Figure 12.2).
3. Enter the `LoanOfficer` for the new role.
4. Click `OK`.
5. Repeat steps 1–4 and select `TaxPreparer` for the `Role` name.

You now need to do two things to use COM+ security declaratively: (1) Add Users to the role, and (2) Specify which entities in the COM+ application: Components, Interfaces, and Methods, that users in the role are allowed to access.

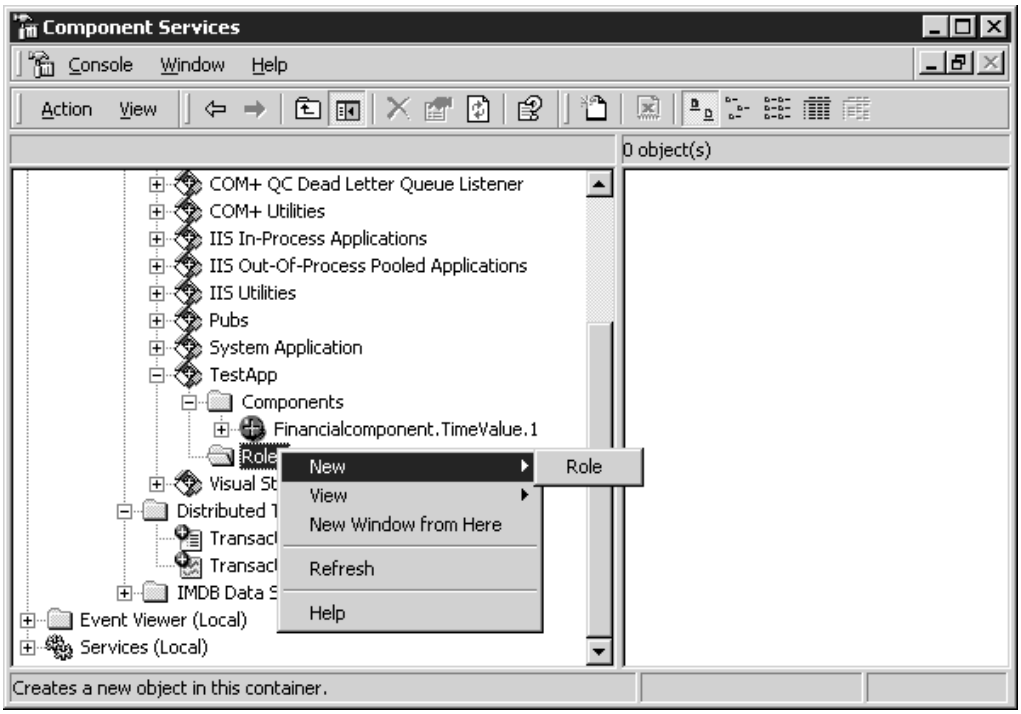



**FIGURE 12.1** Create a New Role.


**FIGURE 12.2** The New Role Dialog.

### ADD USERS TO A ROLE

Perform the following steps to add new users to a role:

1. Right-click on the `Users` folder beneath the `LoanOfficer` role in the `Roles` folder as shown in Figure 12.3.

2. Select `New\Users` from the `Context` menu that appears (the next window that you see depends on whether you are using the Active Directory Service).
3. Select your user name in the list of Users and Click the Add button.
4. Click `OK`.
5. Repeat steps 1 thru 3 using the `TaxPreparer` role.

Remember that by adding a user to a role, you are saying this user is allowed to use all the classes, interfaces, and methods that you will allow this role to access.

### SPECIFY THE ENTITIES EACH ROLE IS ALLOWED TO ACCESS

You need to specify the classes, interfaces, and methods that each role is allowed to access. But before you do this, you have to tell COM+ that you want it to enforce Access Control. By default, COM+'s enforcement of Access Control is turned off. To get COM+ to enforce Access Control, perform the following steps in the Component Services Explorer:

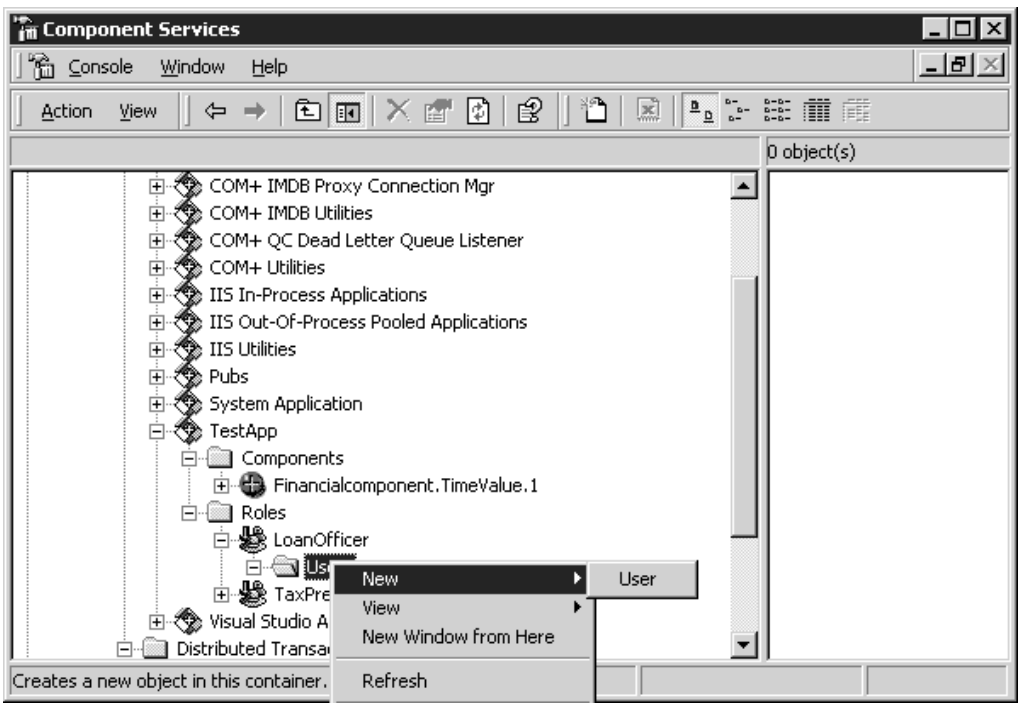



**FIGURE 12.3** Add a User to a Role.

1. Right-click on your COM+ Application (`TestApp`) as shown in Figure 12.4.
2. Select `Properties...` from the `Context` menu (the component properties dialogs appear).
3. Click on the `Security` tab of the component properties dialog (you should see the dialog shown in Figure 12.5).
4. Set the check box that says `Enforce Access checks for this Application`.
5. Click `OK`.

Changes to COM+ security settings do not take effect until the server process is restarted. Make sure the server process has been shutdown by right-clicking on the COM+ Application in the Component Services Explorer and then selecting `Shut down` from the `Context` menu. Now try running the client for the financial component again.

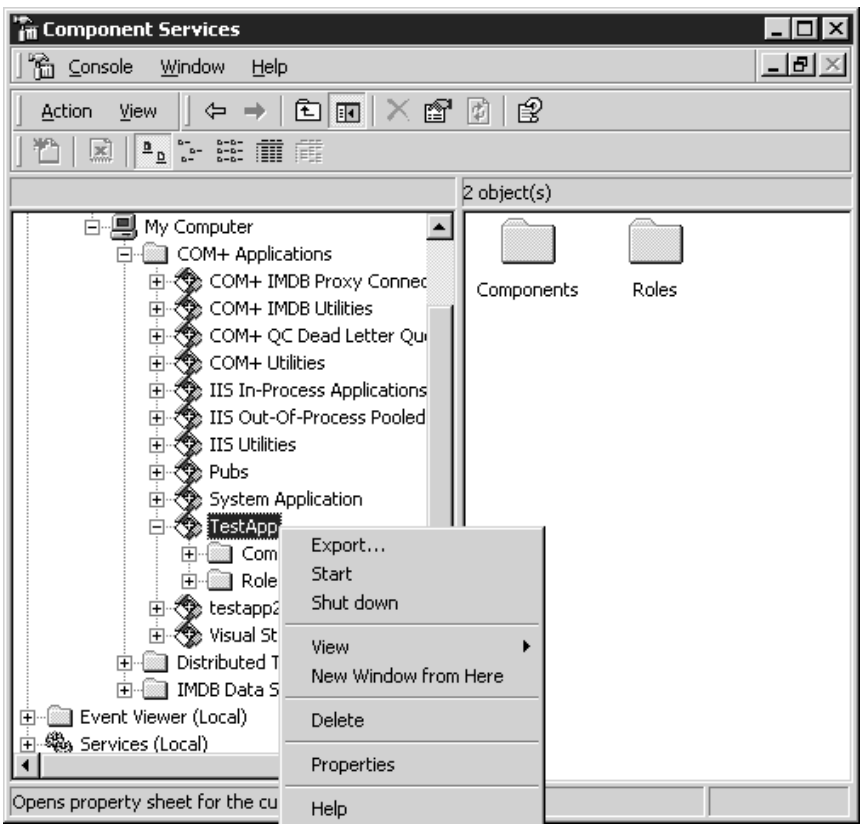



**FIGURE 12.4** Bring Up the Application Properties Dialog.

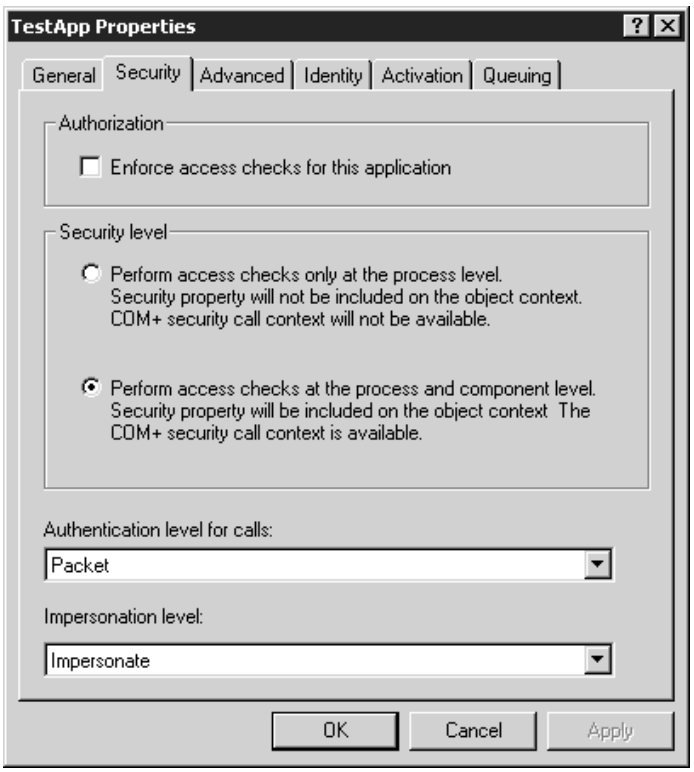



**FIGURE 12.5** The Security Tab of the Application Properties Dialog.


**Any time you change the security settings for a COM+ application, you should always shut down the server process before you test to see the effects of the change. The new changes will not take effect until a new process is created.**

You should see a permission denied error as shown in Figure 12.6.

Understand that COM+ is not actually showing the message box shown in Figure 12.6. COM+ returns the standard, COM, permission denied error. The financial client is catching the error in an error handler and displaying the error description in a message box.

To get this permission denied error to go away, you first need to specify which role(s) are allowed to use the `ITimeValue` interface. As long as the calling user is in one of the roles that are allowed to use the interface, your call will succeed. Perform the following steps to specify that the `LoanOfficer` role is allowed to use the `ITimeValue` interface:

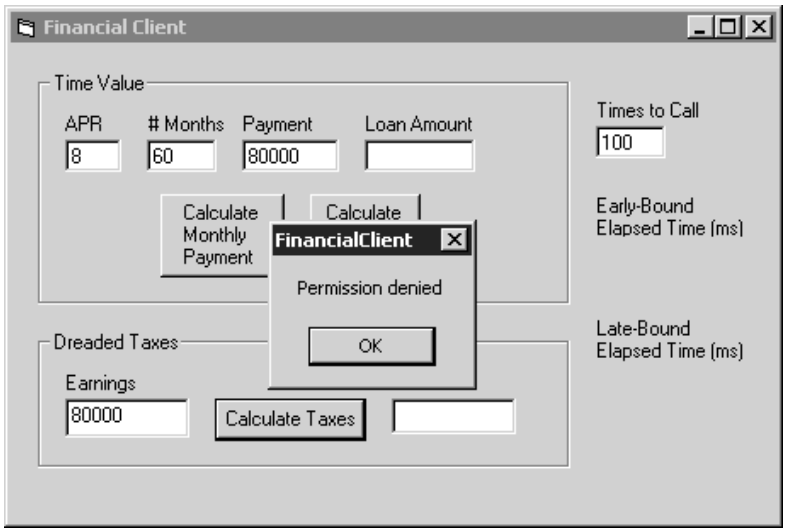



**FIGURE 12.6** COM+ Denying Permission to Use a Component.

1. Right-click on the `ITimeValue` interface beneath the `Interfaces` subfolder as shown in Figure 12.7.
2. Select `Properties...` from the `Context` menu (the interface Properties dialog appears as shown in Figure 12.8).
3. Click the `Security` tab of the interface properties dialog.
4. Set the Checkbox next to the `LoanOfficer` role.
5. Click `OK`.

Make sure you shut down the server process before you try to run the financial client again. When you do run the financial client, you should be able to use the `Calculate Monthly Payment` and `Calculate Loan Amount` buttons at the top of the screen, as long as the current user has been added to the `LoanOfficer` role. These buttons use the `ITimeValue` interface. Now try using the tax calculation functionality. You should once again receive the permission denied error. The tax calculation functionality in the financial client uses the `ITaxCalculation` interface and no roles have been allowed to use the `ITaxCalculation` interface as yet.

Congratulations. You have just successfully implemented fine-grained security without writing a single-line of code. If nothing that I have said before has convinced you of the elegance and simplicity of COM+ and the beauty of attribute-based programming, hopefully this demonstration has. You were able to add new functionality to an existing component without writing a single line of code. I leave it as an exercise for you to add yourself

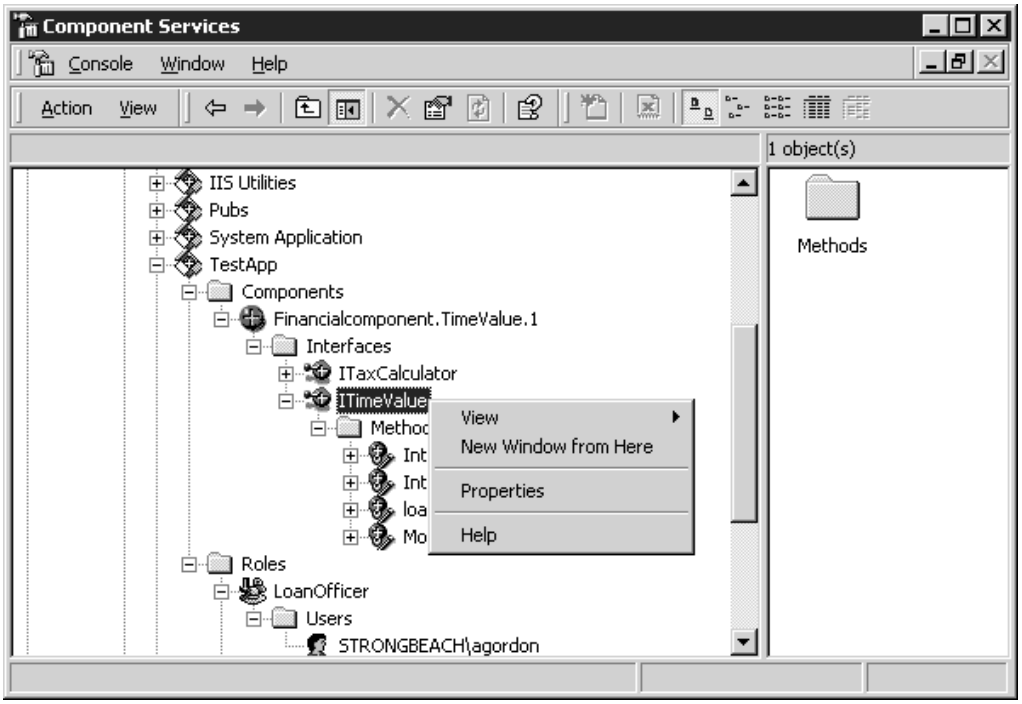



**FIGURE 12.7** Configure Security for the ITimeValue Interface.

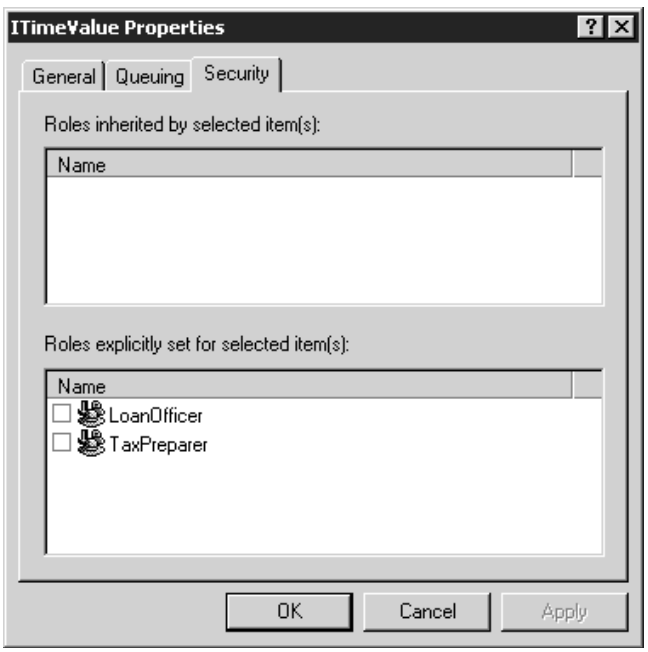



**FIGURE 12.8**

COM+ Interface Properties Dialog.

to the `TaxPreparer` role and then configure the `ITaxCalculation` interface to support the `TaxPreparer` role. After you have done this, you should be able to use the tax calculation functions in the Financial client.

In this scenario, you specified access permissions on a per-interface basis. You can also specify access permissions on a per-method basis. Perform the following steps in the Component Services Explorer to specify access permissions for a particular method:

1. Right-click on a method beneath the `Methods` folder as shown in Figure 12.9.
2. Select `Properties...` from the `Context` menu (the method properties dialog appears as shown in Figure 12.10).
3. Click the `Security` tab of the method properties dialog (shown in Figure 12.10).
4. Set the `CheckBox` for the role you wish to add to this method.
5. Click `OK`.

In this way you can specify different allowed roles individually for each method in an interface.

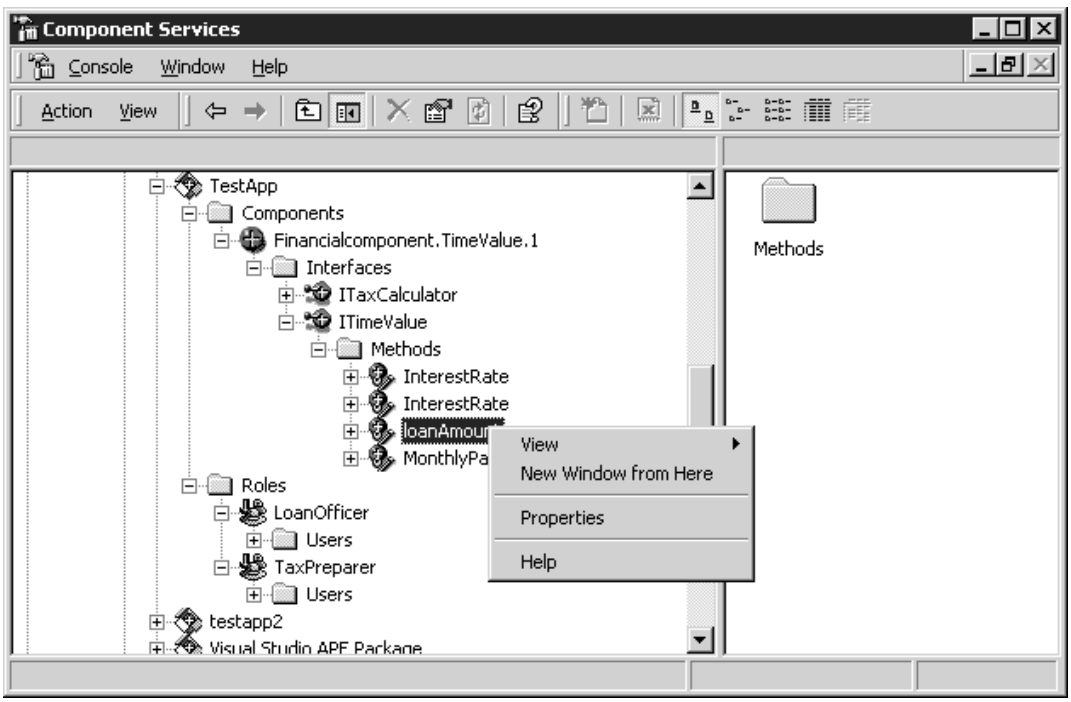



**FIGURE 12.9** Configure Security for the LoanAmount Method.

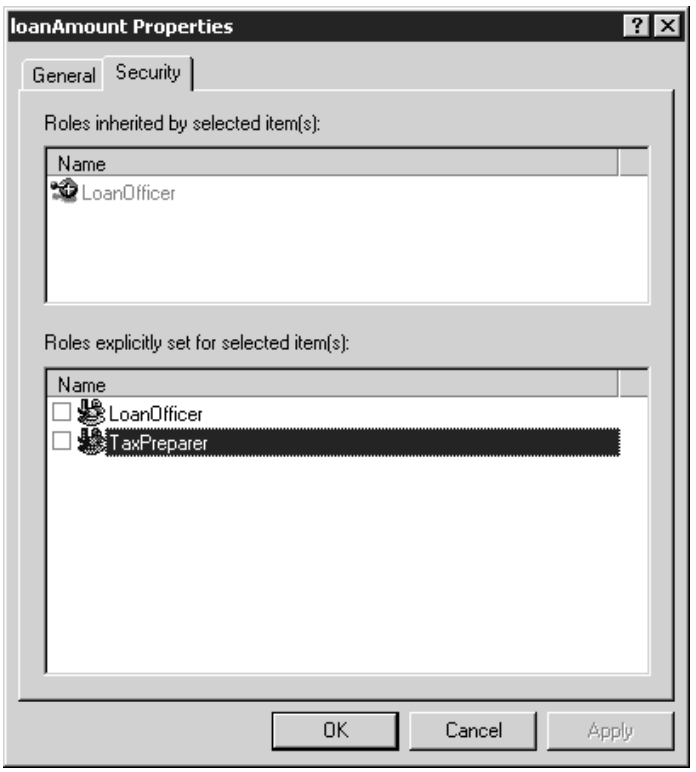



**FIGURE 12.10**

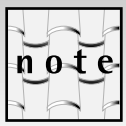
COM+ Method Properties Dialog.

### *Use COM+ Security Programmatically*

There are certain situations where you need to use COM+ security programmatically. Imagine that a bank was using your financial component, and at that bank only managers are allowed to process loans involving amounts over $100,000. Declarative security won't work in this scenario because whether a user is allowed to use a method is a function of the parameters they pass to the method. You have to determine if a method call is allowed at runtime. To add support for this business rule, you first need to add a Manager role to your COM+ Application.


Hopefully by now you should have no problems adding a Manager role to your COM+ application and adding yourself to this role.

After you add the Manager role, you can enforce your security constraint using the `IsCallerInRole` method of the `ISecurityCallContext` interface on the call context.




**Remember from Chapter 11 that the Call Context contains context-related information that is specific to a particular method call.**

With the `IsCallerInRole` method, you can determine if the calling user is in the Manager role and if they are not, you can check to see that the `LoanAmount` they passed to you is less than $100,000. Listing 12.1 shows an updated version of the `MonthlyPayment` method in the `CTimeValue` class that uses COM+ programmatic security.

**LISTING 12.1** *USING COM+ PROGRAMMATIC SECURITY*

```
STDMETHODIMP CTimeValue::MonthlyPayment(short numMonths, double
loanAmount, double *monthlyPayment)
{
   HRESULT hRes, hRetval;
   double monthlyRate, tempVal;
   LPOLESTR errDescription;
   BOOL bIsInRole;
   ISecurityCallContext *pCallContext;
   bstr_t strManagerRole("Manager");
   hRes=CoGetCallContext(IID_ISecurityCallContext,(void **)&pCallContext);
   if (SUCCEEDED(hRes))
   {
      pCallContext->IsCallerInRole(strManagerRole,&bIsInRole);
      if (loanAmount > 100000.0 && !bIsInRole)
      {
         *monthlyPayment=0;
         errDescription=L"The caller is not a manager";
         hRetval=E_CALLERNOTMANAGER;
      }
      else
      {

         monthlyRate=mInterestRate/1200;  // Convert A.P.R. to decimal,
                                                          monthly rate
         tempVal=pow((1+monthlyRate),(double)numMonths);
         *monthlyPayment=loanAmount*monthlyRate*tempVal/(tempVal-1);
```

```
            hRetval=S_OK;
        }
    }
    else
    {
        *monthlyPayment=0;
        errDescription=L"This component must be configured";
        hRetval=E_MUSTBECONFIGURED;
    }
    if (0!=pCallContext)
        pCallContext->Release();
    if (hRetval!=S_OK)
        return Error(errDescription,IID_ITimeValue,hRetval);
    else
        return S_OK;
}
```

I have highlighted the two key lines of code in this function. Notice that you call `CoGetCallContext` to retrieve the `ISecurityCallContext` interface on the call context and then you call `IsCallerInRole` on the interface pointer to test if the user is in the Manager role. If the loan amount is over $100,000 and the user is not in the manager role, then you throw an error. You also throw an error if the component is not configured. To test these changes, you need to add the Manager role to your COM+ application.

After you add the Manager role and make the changes shown above to the financial component, run the application before you add yourself to the Manager role. You should get an error as soon as you try to calculate the monthly payment for a loan greater than $100,000. If you add yourself to the Manager role, you can calculate the monthly payment on any loan. The COM+ security service made it simple to implement programmatic security.

In most cases, you should try to use declarative security; only use programmatic security in situations where declarative security won't work. The example I showed you—where the access permissions for a method are dependent on the values of the parameters that are passed in to the method—is a classic example where declarative security will not work. Declarative security is best because it is easy to use and more flexible. You can change the security behavior of a method without having to alter your source code. With declarative security, it is easy for end-users to change their security configuration. Also, keep in mind that in many situations where it seems that only programmatic security will work, you can still use declarative security if you are willing and able to change your interfaces. For instance, in the scenario that I just sketched out, you could create two methods to calculate loan payments: `MonthlyPayment` and `MonthlyPaymentJumbo`. `MonthlyPayment` always rejects loans over $100,000; `MonthlyPaymentJumbo` accepts any

amount. You can then use declarative security to allow only Managers to call `MonthlyPaymentJumbo`.

## Transactions

One of the key features of COM+ is its support for transactions. But before I can talk about COM+ support for transactions, I need to talk about transactions in general. The word transaction is used in so many different contexts (I'm not talking about COM+ contexts now) that most people have an intuitive sense of what it means. In the field of computer science, here's a precise definition of what a transaction is.

### *What is a Transaction?*

A transaction is a series of operations that have the following properties:

- Atomicity
- Consistency
- Isolation
- Durability

These properties make an easy to remember acronym (ACID). You can understand what these four properties mean by considering what happens when you go to the bank and decide to transfer $500.00 from your savings account to your checking account. There are actually two operations that take place in completing this operation: (1) $500.00 is withdrawn from your savings account, and (2) $500.00 is deposited into your checking account. If both operations succeed, both the bank and you are happy. If both operations fail, neither you nor the bank is necessarily happy, but you can both live with this outcome as long as you can try the operation again later. Now imagine what would happen if the bank's computer failed in the middle of this account transfer operation. First consider the case where the bank's computer failed right after $500.00 was withdrawn from your savings account but before it was deposited into your checking account. You would have lost $500.00. This is not an acceptable outcome. Now consider what would happen if the bank's computer crashed after $500.00 was credited to your checking account, but before the monies were withdrawn from your savings. You would be happy but this is not acceptable to the bank. Both operations must succeed or they must both fail. This is what *Atomicity* means. *Consistency*, in this example, means the amount deposited into your checking account should be the same as the amount withdrawn from your savings account. Consistency is enforced by application logic with the help of a DBMS and/or a Transaction Processing (TP) Monitor.


A TP Monitor is a piece of software that can, among other things, coordinate a distributed transaction involving multiple resources. As you will learn shortly, COM+ contains most of the functionality of a TP Monitor.

*Isolation* means that a separate transaction that is executing concurrently with your account transfer should not see an invalid intermediate state such as where the $500.00 has been withdrawn from savings, but has not yet been deposited into checking. Isolation is usually implemented using locking. The *Durable* property of a transaction means that after the transaction is committed, the updates made by the transaction should never be lost. A system crash, network failure, or even someone inadvertently pulling the power cord, should not cause updates to be lost. Most databases (and other types of transactional resources) implement durability by first writing all intended changes to a journal file in durable storage. Once the journal file is written the database has a permanent record of the changes that need to be made to the system. The database can still apply the changes at a later time, even if there is a catastrophic failure while the resource was being updated.

Most Database Management Systems (DBMS) have built-in support for the Begin, Commit, and Rollback transaction-handling primitives. The basic steps of using these transaction primitives are shown in the following psuedo-code:

```
try
{
   Transaction.Begin
   Withdraw $500.00 from savings account
   deposit $500.00 dollars into checking account
   Transaction.Commit
}
catch (Exception)
{
// if anything goes wrong, rollback the entire operation
   Transaction.Rollback
}
```

The Begin operation starts a transaction. When all the operations in the transaction are complete, you can call the `Commit` function to commit the transaction. If the transaction fails at any time, you can call the `Rollback` function, which undoes everything since the call to `Begin`. You can make several updates to the database, but none of these updates are visible to anyone outside the transaction until the `Commit` function is called. The `Commit` function applys all the updates in an atomic step. Moreover, once the `Commit` function succeeds, you are guaranteed that the

operation will succeed even if someone pulls a power cord. The code just shown also emphasizes the time-saving aspect of transactions. Without transactions you, the developer, would have to write code to undo a partially complete set of operations if one of the operations fails. With transactions, you push the work of undoing the effects of a partial failure onto the database. If anything goes wrong, you simply rollback the transaction.

### *Distributed Transactions*

Using the transaction management functions that are built into your DBMS will work okay as long as all of the information is stored in a single database. Unfortunately, most enterprises don't store all their important information in a single database. In many cases, the information is spread among many databases. Take an online retailer as an example. A database of customer account information might be stored in an Oracle database at the corporate office. But many e-commerce businesses do not fill their orders themselves; they simply run the Web site and advertise. One or more partners actually fill the orders. In this situation, when the online retailer receives an order, it must be able to send the order information to its partners. The mechanism used to send this information could be to write directly to the partner's SQL Server database, or they might send a message to their partner using a message queuing product like Microsoft Message Queue (MSMQ). In either case, you need the placing of the order to be filled and the debiting of the customer's account to be done in an ACID way. The problem in this case is that you have multiple, distributed *resource managers* involved in the transaction.


A resource manager is a term from the field of transaction processing. It is a system that can provide ACID operations on the objects it implements. The canonical resource manager is an RDBMS such as Microsoft SQL Server. But MSMQ is also a resource manager and now, thanks to the COM+ Compensating Resource Manager (CRM) service, you can easily create your own custom resource managers.

Either all of the servers must commit their part of the transaction or, if any of the resource managers are unable to commit their part of the transaction, all of them must rollback.

### *2-Phase Commit Protocol*

The key to implementing distributed transactions is the 2-phase commit protocol. In this protocol, the activity of the resource managers must be controlled by a separate piece of software that is sometimes called a transaction manager or a transaction coordinator. The steps in this protocol are shown below.

- An application invokes the commit method in the transaction coordinator.
- The transaction coordinator contacts each resource manager involved in the transaction and tells it to prepare to commit the transaction (this is the beginning of phase 1).
- To respond in the affirmative to the prepare phase, a resource manager must put itself into a state where it can guarantee to the transaction coordinator that it will commit the transaction if told to do so, or rollback the transaction if told to do so. Most resource managers will write a journal file (or the equivalent) with its intended changes to durable storage. If the resource manager is unable to prepare the transaction, it replys to the transaction coordinator with a negative response.
- The transaction coordinator collects all of the responses from the resource managers.
- In phase 2, the transaction coordinator informs each resource manager of the outcome of the transaction. If any of the resource managers responded in the negative, then the transaction coordinator sends a rollback command to all of the resource managers involved in the transaction. If they all responded in the affirmative, then the resource managers are instructed to go ahead and commit the transaction. Once the resource managers are told to commit, the transaction cannot fail after that. By responding in the affirmative to phase 1, each resource manager was guaranteeing that the transaction would not fail if it were told to commit later.

The 2-phase commit protocol is actually ubiquitous. If you are married, or have attended a wedding, you have seen the 2-phase commit protocol in action. In a wedding ceremony, the clergyman performing the ceremony is the transaction coordinator and the two resource managers are the bride and groom. In the prepare phase, the clergyman asks each resource manager to prepare to commit the transaction: he asks each person if they take the other to be their lawfully wedded spouse. Assuming each responds in the affirmative, he then turns to the crowd (who, in some sense, are a third resource manager) and asks if anyone objects to the marriage. If either the bride or the groom gets cold feet, or someone in the crowd objects, the clergyman can rollback the transaction, which in this case means to cancel the wedding. Assuming both the bride and groom say yes to the marriage and everyone in the crowd stays silent, the ceremony moves to the commit phase where the happy couple exchange rings, the clergyman pronounces them man and wife, and the couple kisses to signify that the transaction (marriage) has been committed.

If you don't like my marriage analogy and you prefer to look at pictures, Figures 12.11 and 12.12 show the 2-phase commit protocol as a pair of sequence diagrams. Figure 12.11 shows the sequence diagram where the

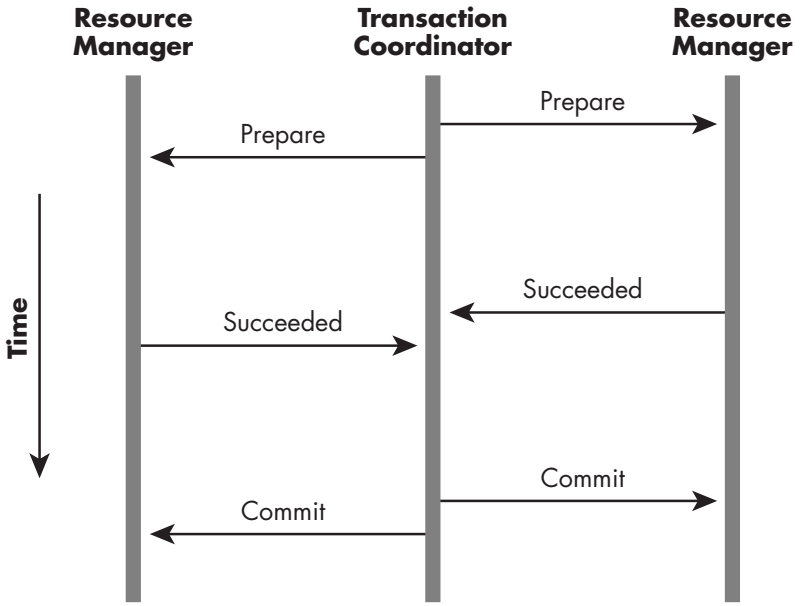



**FIGURE 12.11** The 2-Phase Commit Protocol When the Transaction Commits.

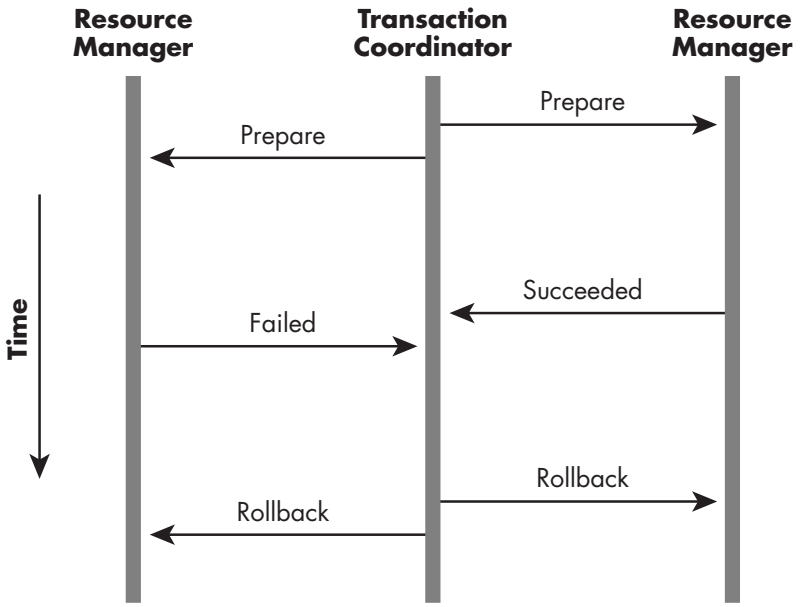



**FIGURE 12.12** The 2-Phase Commit Protocol When the Transaction Is Rolled Back.

transaction succeeds (commits). Figure 12.12 shows the 2-phase commit protocol when one of the resource managers is unable to commit for some reason.

### *Transactions and COM+*

Using COM+, you can add support for distributed transactions to your COM components with the barest minimum of code. The process is simple.

1. Configure your object to run within COM+.
2. Set the Transaction Attribute of the Object to Requires a Transaction, Requires a New Transaction, or Supports Transaction.
3. Call `SetComplete` on the object context when you are ready to commit a transaction.
4. Call `SetAbort` on the object context to indicate when a transaction must be rolled back.

Table 12.1 summarizes the meaning of all the possible values that may be set for the transaction attribute of an COM+ object.

**TABLE 12.1** Transaction Attributes

| Transaction Attribute | Description |
|---|---|
| Requires a Transaction | This object must run within a transaction. COM+ only starts a new transaction for this object if some other object running in the same context has not already started a transaction. |
| Requires a New Transaction | This object must run within a transaction. COM+ starts a new transaction as soon as the object is created. |
| Supports Transactions | This object should only run within a transaction if its creator has already created one. If its creator has not started a transaction this object runs without a transaction. |
| Does Not Support Transactions | This object does not require a transaction and COM+ will not run this object within a transaction (the default). |
| Disabled | This object ignores completely the value of the transaction attribute when determining context placement. This setting makes your configured component behave like an unconfigured one. |

You can configure the transaction attribute on a per-class basis in the Component Services Explorer by performing the following steps:

1. Right-click on a component in the `Components` folder of your application.
2. Select `Properties...` from the `Context` Menu (the Component Properties dialog appears as shown in Figure 12.13).

3. Click the `Transactions` tab (shown in Figure 12.13).
4. Select one of the transaction settings.
5. Click `OK`.

That's all there is to it. You never have to code the typical `Begin`, `Commit`, `Rollback` logic again. When you start using an object that is configured to use transactions, COM+ starts a transaction. Remember COM+ can do this because it uses Interception. The COM+ runtime receives client requests before the actual object does and it can perform actions on behalf of the object.

**START A TRANSACTION**

After you have configured a COM+ component (class) to use transactions, the COM+ runtime reads the transaction attribute when it creates an object. It then contacts COM+'s transaction coordinator, which is called the Distributed Transaction Coordinator or (DTC). It asks the DTC to start a transaction for the object. The DTC starts a transaction for the object and returns the GUID for the transaction (the DTC assigns a GUID to each transaction). The GUID for

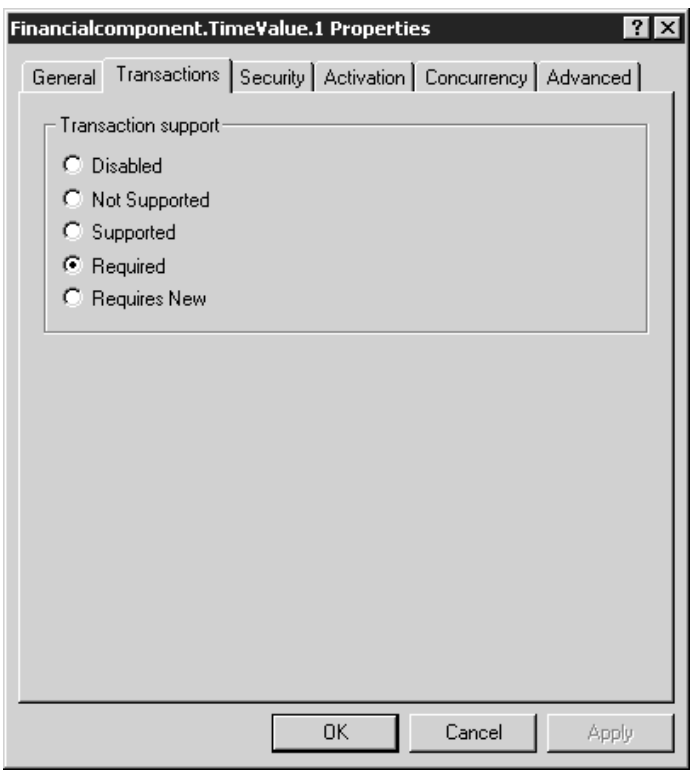



**FIGURE 12.13**

The Transaction Tab of the Components Properties Dialog.

the transaction is carried with the object in its context. You can determine if your object currently is running within a transaction and determine the GUID of its transaction by calling `CoGetObjectContext` and requesting the interface `IObjectContextInfo`. You can then use the `IsInTransaction` and `GetTransactionId` methods of `IObjectContextInfo` to determine if the object is in a transaction and retrieve the GUID of the transaction. The following code shows how this is done:

```
GUID aGUID;
BOOL bIsInTransaction;
IObjectContextInfo *pContextInfo;
hRes=CoGetObjectContext(IID_IObjectContextInfo,(void **)&pContextInfo);
if (SUCCEEDED(hRes))
{
   bIsInTransaction=pContextInfo->IsInTransaction();
   if (bIsInTransaction)
      pContextInfo->GetTransactionId(&aGUID);
   pContextInfo->Release();
}
```

The object that actually causes a new transaction to be created is called the root object of the transaction. The root object is important because the COM+ runtime won't attempt to commit the transaction until the root object is deactivated. Other objects can enlist in the transaction after the transaction has been created and these other objects can vote as to whether the transaction can commit. But, the root object (actually the deactivation of the root object) determines *when* the transaction commits.

When a COM+ object that has enlisted in a transaction makes a connection to a COM+ Resource Manager (in most cases this is a COM+ aware relational database), the Resource Manager also enlists in the transaction. The resource manager contacts the DTC and lets the DTC know that it is now a participant in the transaction. If the object connects to multiple resource managers, for instance, an Oracle database containing customer information, and a Microsoft SQL Server database containing order fulfillment information, each of these resource managers enlist in the transaction.


**Most of the major database vendor's products are COM+ aware (or will be by the time COM+ is released). All versions of Microsoft SQL Server after 6.5 are COM+ aware, as are Oracle 7.3 and 8.0, Sybase SQL Server, IBM DB2, Informix, and CA Ingress. Other products, such as Microsoft Message Queue (MSMQ), are also COM+ Resource Managers.**

As you perform operations on one or more resource managers, they buffer the operations maintaining a journal of your actions. When you have

completed all of the work and are happy with the current state of the transaction, all you have to do is call `SetComplete` on the Object Context. If at any time you receive an error you simply call `SetAbort`. Listing 12.2 contains example code showing how this is done.

**LISTING 12.2** *USING COM+ TRANSACTION WITH IOBJECTCONTEXT*

```
HRESULT hRes;
   IObjectContext *pObjectContext;
hRes=CoGetObjectContext(IID_IObjectContext,(void**)&pObjectContext);

   try
   {
   // Perform work here;

      pObjectContext->SetComplete();
   }
   catch(exception err)
   {
      pObjectContext->SetAbort();
   }
   pObjectContext->Release();
```

When you call `SetComplete`, you are voting to commit the transaction. Once the root object of the transaction indicates that it is ready to commit (by calling `SetComplete`), the DTC goes to work and using the 2-phase commit protocol it commits the transaction on all of the resource managers involved in the transaction. If any of the resource managers is unable to commit the transaction, the DTC rolls the entire transaction back.

### TRANSACTIONS INVOLVING MULTIPLE OBJECTS

The case just described is where you have one COM+ object executing a transaction against 2 or more resource managers. In many cases, a transaction involves multiple COM+ objects. A good example is for an online book retailer where you might have an Order COM+ object that is used as a transaction root. This Order object may create and use a Customer COM+ object and a Book COM+ object in its processing of an order as shown in Figure 12.14.

When a configured COM+ instantiates another configured COM+ object, the new object can do one of three things with its creator's transaction: (1) it can share its creator's transaction, (2) it can create its own transaction, (3) it can not run within a transaction at all. How the new object behaves depends on the setting for the transaction attribute on both the creating and created objects. Table 12.2 looks at the example of an Order

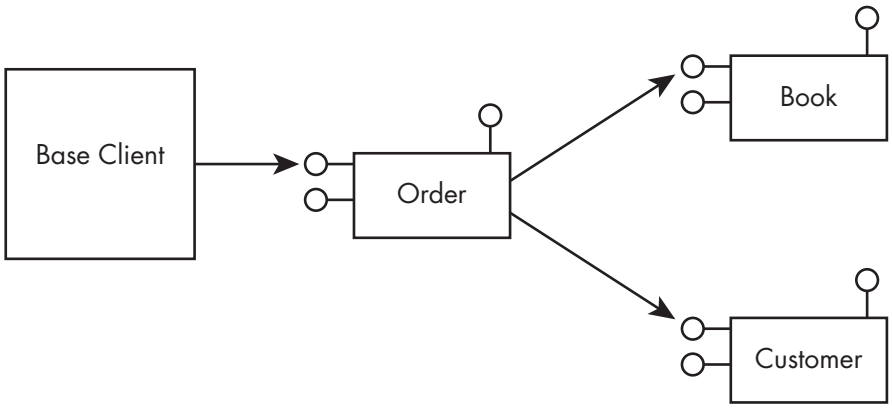



**FIGURE 12.14** Multiple Objects in a COM+ Transaction.

object instantiating a Customer object and shows the transaction state for each possible transaction setting on each object (I left the disabled setting out of the table).

**TABLE 12.2** Transaction Attributes

| Order Transaction Attribute | Customer Transaction Attribute | The Order Object will run in a transaction | The Customer Object will run in a transaction | The Customer and Order Objects will share the same transaction |
|---|---|---|---|---|
| Requires a New Transaction | Requires a New Transaction | Yes | Yes | No |
| Requires a New Transaction | Requires a Transaction | Yes | Yes | Yes |
| Requires a New Transaction | Supports Transactions | Yes | Yes | Yes |
| Requires a New Transaction | Does not Support Transactions | Yes | No | N/A |
| Requires a Transaction | Requires a New Transaction | Yes | Yes | No |
| Requires a Transaction | Requires a Transaction | Yes | Yes | Yes |
| Requires a Transaction | Supports Transactions | Yes | Yes | Yes |

| Order Transaction Attribute | Customer Transaction Attribute | The Order Object will run in a transaction | The Customer Object will run in a transaction | The Customer and Order Objects will share the same transaction |
|---|---|---|---|---|
| Requires a Transaction | Does Not Support Transactions | Yes | No | N/A |
| Supports Transactions | Requires a New Transaction | Only if the object that activated the Order has a transaction. | Yes | No |
| Supports Transactions | Requires a Transaction | Only if the object that activated the Order has a transaction. | Yes | If the Order object has a transaction, the Customer will share it. If not, the Customer will have its own transaction. |
| Supports Transactions | Supports Transactions | Only if the object that activated the Order has a transaction. | Only if the object that activated the Order has a transaction. | If the Order object has a transaction, the Customer will share it. If not, the Customer will have its own transaction. |
| Supports Transactions | Does Not Support Transactions | Only if the object that activated the Order has a transaction. | No | N/A |
| Does Not Support Transactions | Requires a New Transaction | No | Yes | N/A |
| Does Not Support Transactions | Requires a Transaction | No | Yes | N/A |
| Does Not Support Transactions | Supports Transactions | No | No | N/A |
| Does Not Support Transactions | Does Not Support Transactions | No | No | N/A |

If the Order object and the Customer object each have their own transaction, then they are each on their own as it regards committing their transaction. The operations performed by the Order object can commit while the Customer's transaction can abort. In some cases, this may be what you want, in most cases, this is *not* what you want. In most cases, you want the Order object and the Customer object to share the same transaction. If the Order object and the Customer object share the same transaction, then the outcome of the transaction depends on how each of the objects *vote* on whether the transaction should commit. An object can vote to commit a transaction by calling the `SetComplete` method on the `IObjectContext` interface of its object context. An object can vote to abort (rollback) a transaction by calling `SetAbort` on the `IObjectContext` interface of its object context. If both objects vote to commit the transaction, the transaction commits; if *either* object votes to rollback the transaction, then the transaction is rolled back. It's just like a wedding. If the bride says *no* when asked if she takes the groom to be her husband, it really doesn't matter if the groom had already said *yes*; the entire ceremony is aborted. This last point is an important one because it has another implication: once the groom gives his consent to the wedding, the outcome of the ceremony is still in doubt until the bride gives her consent. The same is true for COM+ objects. Once an object calls `SetComplete`, it doesn't know if the transaction will commit or not. The object is only giving its acceptance to the commitment of the transaction. This has an important implication for the management of the state of a transactional object. A transactional COM+ object should not assume that the state it has when it calls `SetComplete` will be valid after the transaction is committed. For instance, if the Customer object debits a customer's account in a database, it should not maintain state that contains the customer's new account balance; it should refresh itself with the new balance from the database after the transaction is committed. Even if the Customer object was pleased with its updates, the Book object could still rollback the transaction (perhaps because the book is no longer in print), and if it did, the Customer object is now in an invalid state. To prevent problems like this from happening, COM+ deactivates all the objects involved in a transaction when the transaction commits. This forces you to reinitialize your objects after each transaction. Deactivating objects like this also helps scalability, but the main reason it is done is to ensure the correctness of a transaction.

Deactivating an object means different things depending on whether object pooling is turned on (I talk about object pooling shortly). If object pooling is not turned on, then the deactivated object is actually destroyed. If object pooling is turned on, the deactivated object is returned to the object pool where another client can use it later.

## MANAGING STATE IN COM+ OBJECTS

There is a big problem with this deactivate-on-transaction-commit approach. The definition of an object (in the computer science sense) is that it is an entity that has behavior and *state*. If the state of a transactional COM+ object is lost as soon as you commit a transaction, it makes it difficult to maintain the association between behavior and state that is the essence of an object. The real question is: how do you keep track of the state of your transactional COM+ objects?

You have a number of choices as to maintaining object state: You can make your object stateless, you can cache your object's state in the shared property manager, or you can persist the object's state into a database or some other durable storage.

**STATELESS OBJECTS** • Stateless objects are objects that do not maintain any member variables. This means that all of the information the object requires to execute a method must be stored somewhere (usually on the client), then passed into each method, or retrieved from a database or some other persistent storage. Some people have the misconception that to use COM+ (or MTS), your COM objects *must* be stateless. This is not true. Using COM+ requires you to think carefully about how you manage state in your objects. Building stateless objects is only one way to use COM+.

**THE SHARED PROPERTY MANAGER** • Another way to manage state in a COM+ object is to store the object's state in the Shared Property Manager, which is affectionately called the SPAM. The SPAM is an in-memory cache that is maintained within the COM+ surrogate process. It is essentially an indexed collection. The SPAM is good for maintaining process-wide, transient data (that's a fancy name for global variables). Because the SPAM runs in the surrogate process, all data in the SPAM is lost when the surrogate process shuts down. By default, that is three minutes after there has been no activity. You can alter this setting in the Component Services Explorer.

You can place data of any type in the SPAM and index it via a string key. When you wish to fetch the data later, you ask for it by its key. Listing 12.3 shows how to maintain a global counter in the SPAM. Each time this code is run it increments the counter. For clarity and brevity, I've omitted error checking from this code.

**LISTING 12.3** *USING THE SPAM*

```
1. STDMETHODIMP CTestObject::TestSpam(long *counter)
2. {
3.     LONG lIsolationMode = LockMethod;
4.     LONG lReleaseMode = Process;
5.     _variant_t vtNextNumber;
6.     HRESULT hRes;
7.     ISharedPropertyGroupManager* pPropGrpMgr;
8.     ISharedPropertyGroup* pPropGrp;
9.     ISharedProperty* pPropNextNumber;
10.    VARIANT_BOOL bExists;
11.
12.    hRes=CoCreateInstance(CLSID_SharedPropertyGroupManager,NULL,
13.          CLSCTX_ALL,IID_ISharedPropertyGroupManager,
           (void**)&pPropGrpMgr);
14.
15.    hRes=pPropGrpMgr->CreatePropertyGroup(L"SharedCounters",
16.       &lIsolationMode, &lReleaseMode, &bExists,&pPropGrp);
17.    pPropGrpMgr->Release();
18.
19.    pPropGrp->CreateProperty(L"NextNumber",&bExists,&pPropNextNumber);
20.    pPropGrp->Release();
21.
22.    vtNextNumber.vt=VT_I4;
23.    hRes = pPropNextNumber->get_Value(&vtNextNumber);
24.    *counter=vtNextNumber.lVal++;  // Increment the counter
25.    hRes = pPropNextNumber->put_Value(vtNextNumber);
26.    pPropNextNumber->Release();
27.    return S_OK;
28.}
```

In this code, you call the `CoCreateInstance` method to obtain the `ISharedPropertyGroupManager` interface on a `SharedPropertyManager` object on lines 12 and 13. Next, create a property group called `SharedCounters` using the `CreatePropertyGroup` method of `ISharedPropertyGroupManager` as shown on lines 15 and 16. This method opens the property group if it already exists, or it creates the property group if it does not. A property group is just a collection of properties. Next, you can attempt to create a property called `NextNumber` using the `CreateProperty` member function of the property group as shown on line 19. Once again, this method returns an existing property if it exists already or creates a property with the name if it does not. Finally, on lines 22–26 you get the value associated with the property (if the property is being created for the first time, its initial value is

zero). You then increment the value and store the new value back into the property. The helpful thing about using the SPAM to maintain data like this is that it is global to the server not the client. This means that all clients see the same value for this property. Remember that once the server process shuts down, all data in the SPAM is lost.

**STORING STATE IN PERSISTENT STORAGE** • Your final choice for dealing with state in COM+ objects is to store the object state in persistent storage: a file or a database. When your object is deactivated, you can write its state to persistent storage and later, when an object is activated again, it can read its state back from the database. The `IObjectControl` interface comes in handy when implementing this functionality. If a COM+ object implements the `IObjectControl` interface, COM+ calls the `Deactivate` method on `IObjectControl` when it deactivates the object and it calls the `Activate` method on `IObjectControl` when it activates the object. You can put logic in the `Deactivate` method to write state to a database or file and then read the data back in from the database or file when the `Activate` method is called. I talk more about `IObjectControl` and you will see how to implement it when I discuss object pooling.

**STATEFUL, TRANSACTIONAL OBJECTS: DON'T DO IT!** • You can create stateful, transactional objects in COM+ if you really want to. As long as you never indicate to COM+ that the object is done with its work (basically, that means you don't call `SetComplete` or `SetAbort`), an object is not deactivated until the base client deactivates the object by releasing its reference. Of course, an object designed this way does not commit any of its transactions until the client releases the object. This fact alone makes it impractical in most situations to design your objects this way. But there are other problems, too: shared resources used by the object (database connections and memory) are also not reclaimed until the client releases its references to the object, ruining scalability. In short, trying to create stateful, transactional COM+ objects is a bad idea.

### DEACTIVATE-ON-RETURN BIT AND THE TRANSACTION-VOTE BIT

So far, I have simplified my discussion of how a transaction is controlled and committed so you would understand the basics. Now that you have a better understanding of the relationship between a transaction and the lifetime of an object, I can tell you that the situation is actually slightly more complicated (for good reasons) than what you have seen to this point. To understand how a transaction is controlled in COM+, you must understand the role of the deactivate-on-return and transaction-vote bits in the object context.

Each object's context contains two bits: a transaction-vote bit that indicates if the object is happy with the transaction as it currently exists (if it would vote to commit the transaction right now), and a deactivate-on-return

bit that indicates if the object has completed its work and can be deactivated (remember COM+ must deactivate an object when it commits a transaction). The transaction-vote bit is also sometimes called the Happy bit and the deactivate-on-return bit is sometimes called the Done bit.


**Under MTS (which COM+ replaces), the transaction-vote and deactivate-on-return bits were always called the Happy and Done bits, respectively. Under COM+, the terminology has officially changed. Transaction-vote and deactivate-on-return are the new, more accurate names for these bits. Throughout the rest of this chapter, I stick with the official terminology and use transaction-vote and deactivate-on-return for these bit names.**

A COM+ object can change its transaction-vote bit any time until it is deactivated. Once an object is deactivated, the state of its transaction-vote bit at that time becomes its vote as to whether its transaction should succeed or fail. If the transaction-vote bit is set, when an object is deactivated, the object is voting to commit the transaction; if the transaction-vote bit is cleared, the object is voting to abort the transaction. Continuing with the marriage analogy, setting its transaction-vote bit is a COM+ object's way of saying I do. Clearing its transaction-vote bit is a COM+ object's way of saying I don't.

Under COM+, an object can be deactivated in two ways: (1) the client can release its reference to the object (call Release on an interface), or (2) an object can set its deactivate-on-return bit. If a COM+ sets its deactivate-on-return bit inside a method, the COM+ runtime deactivates the object when the method returns. Because the COM+ runtime won't actually attempt to commit a transaction until the object that created the transaction (the root object) is deactivated, setting the deactivate-on-return bit also has the effect of telling the COM+ runtime it can commit the transaction anytime it is ready.

Now that you understand these bits you can understand exactly what happens when a COM+ object calls `SetComplete` on the `IObjectContext` interface of its object context inside of a method. `SetComplete` sets both the transaction-vote and deactivate-on-return bits to true. This has the effect of telling COM+ that you are happy with the transaction and it can go ahead and commit it any time it is ready. Calling `SetAbort` on the `IObjectContext` interface of the object context clears the transaction-vote bit and sets the deactivate-on-return bit. This causes the object to be deactivated with its transaction-vote bit cleared and causes the transaction to abort.

So far, you have only looked at transactions that last for a single method. In these cases, the method calls `SetComplete` or `SetAbort` before it returns. What happens if you want to create a transaction that spans multiple method calls on an Object? Let's say you're designing an Order object. The use case for the object is as follows: the user first fills in the Master information for the order (customer name, delivery address,

credit card information, and so on). Next, the user fills in the line items for the order and finally submits the order. The `Submit` method performs some required validity checks on the order and then submits the order for processing. If you cannot submit the order for any reason, all information associated with the order should be removed from the database. A sequence diagram for this use case is as shown in Figure 12.15.

In this use case, you don't want the transaction to commit until the client calls the `Submit` method. So the sequence of events for a transaction is as follows:

1. The User creates an Order object, which starts a transaction.
2. They then call `AddLineItem` one or more times within the same transaction.
3. The user calls `Submit` to submit the order and commit the transaction.

Obviously, you don't want to call `SetComplete` in `SetMasterInfo` because that commits the transaction. You also don't want to call `SetComplete` in `AddLineItem` because that also commits the transaction.

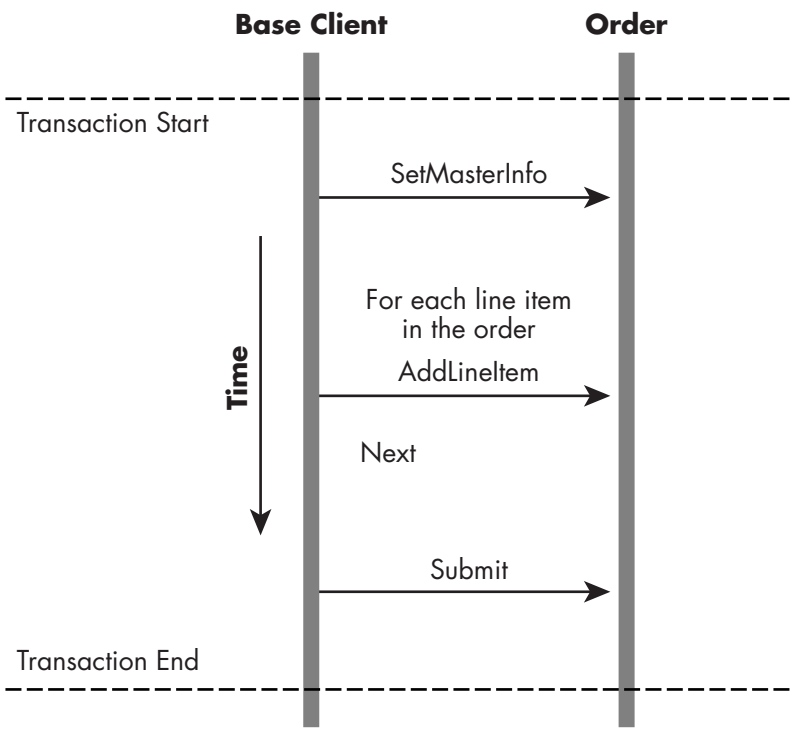



**FIGURE 12.15** A Sequence Diagram for Placing an Order.

You only want to call `SetComplete` in the `Submit` method. Moreover, if the `Order` object is deactivated before the `Submit` method is called, you want the transaction to automatically abort. You can implement this functionality using the `DisableCommit` method on the `IObjectContext` interface of the object context. `DisableCommit` clears both the transaction-vote bit and the deactivate-on-return bit. The COM+ runtime does not attempt to deactivate the object and if the base client releases its last reference to the object (which causes the object to be deactivated), the fact that the transaction-vote bit is false causes the transaction to be aborted. In the scenario sketched out in Figure 12.15, you should call `DisableCommit` in `SetMasterInfo` and `AddLineItem` and then call `SetComplete` in the `Submit` method.

If you create a table showing all the possible states of the deactivate-on-return and transaction-vote bits as shown in Table 12.3, you will see that we have explored the first three of the four possible states.

**TABLE 12.3** Possible States for the Deactive-On-Return and Transaction-Vote Bits in the Object Context

| Deactivate-on-return Bit | Transaction-Vote Bit | IObjectContext Method | Object Deactivated on Method Return | OK to Commit Transaction |
|---|---|---|---|---|
| On | Off | SetAbort | Yes | No |
| On | On | SetComplete | Yes | Yes |
| Off | Off | DisableCommit | No | No |
| Off | On | EnableCommit | No | Yes |

What about the last state? This is where the deactivate-on-return bit is cleared, so the object is not deactivated when the current method returns, but the transaction-vote bit is set, so the transaction is committed if the object is deactivated by the base client releasing its reference. You can put the deactivate-on-return and transaction-vote bits in this state by calling `EnableCommit` on the `IObjectContext` interface. An object should call `EnableCommit` if it is *not* yet done with all the work it would *like* to do, but if the object is deactivated, its transaction should be committed anyway. `EnableCommit` is the default action if a method in a COM+ object returns from a method without calling any methods on `IObjectContext`.

With Microsoft Transaction Server on NT 4.0, the only way you could manipulate the transaction-vote and deactivate-on-return bits was through the `IObjectContext` interface. Under COM+, the object context supports a number of interfaces including `IContextState`, which allows you to manipulate the transaction-vote and deactivate-on-return bits explicitly. `IContextState` contains the four methods shown in Table 12.4.

**TABLE 12.4** The Methods in the IContextState Interface

| Method Name | Description |
|---|---|
| SetDeactivateOnReturn | Sets the deactivate-on-return bit to true or false |
| GetDeactivateOnReturn | Gets the Done bit |
| SetMyTransactionVote | Sets the transaction-vote bit to true or false |
| GetMyTransactionVote | Gets the transaction-vote bit |

Using `IContextState`, you can write the following code to either commit or abort a single-method transaction.

**LISTING 12.4** *USING A COM+ TRANSACTION WITH ICONTEXTSTATE*

```
HRESULT hRes;
IContextState *pContextState;
hRes= CoGetObjectContext(IID_IContextState,(void **)&pContextState);

try
{
// Perform work here;

   pContextState->SetMyTransactionVote(TxCommit);
}
catch(exception err)
{
   pContextState->SetMyTransactionVote(TxAbort);
}
ContextState->SetDeactivateOnReturn(VARIANT_TRUE);
pContextState->Release();
```

If you have a transaction that spans multiple methods, you can call `SetMyTransactionVote` in each method and call `SetDeactivateOnReturn` with a `VARIANT_FALSE` parameter to prevent the object from deactivating. Then, in the method where you commit the transaction, you can call `SetDeactivateOnReturn` and pass in `VARIANT_TRUE`.

Using the `IContextState` interface is equivalent to using `IObjectContext`. I'm not sure which is the preferred way to manipulate a transaction. An article by Don Box in the September 1999 edition of *MSJ* stated that the use of `IObjectContext` is deprecated under Windows 2000, but almost all of the COM+ examples on Microsoft's Web site use `IObjectContext`. I use `IContextState` in the COM+ example you build in the next chapter.

I should make a couple of points before leaving the subject of transactions. First, a transaction is doomed once one of the objects involved in the

transaction returns from a method with its deactivate-on-return bit set to true and its transaction-vote bit set to false (an object can do this explicitly using `IContextState` or by calling `SetAbort` on `IObjectContext`). You might as well set up your error handling so the error is immediately returned to the calling object (if there is one) so it can also abort. It's like the groom saying no when asked if he takes the bride to be his wife. Once he says no, you might as well call off the wedding immediately; there's no point in asking the bride if she takes him to be her husband. There's an example of how to setup your error handling in Chapter 13 when you build a complete COM+ example.

Second, you can change the default behavior that COM+ has in regard to transaction management. You can do this on a method-by-method basis by setting the Auto-Done bit to true. Perform the following steps in the Component Services Explorer to set the Auto-Done bit to true for a method:

1. Right-click on the method beneath the `Methods` folder under the `Components` folder in your COM+ Application (see Figure 12.16).

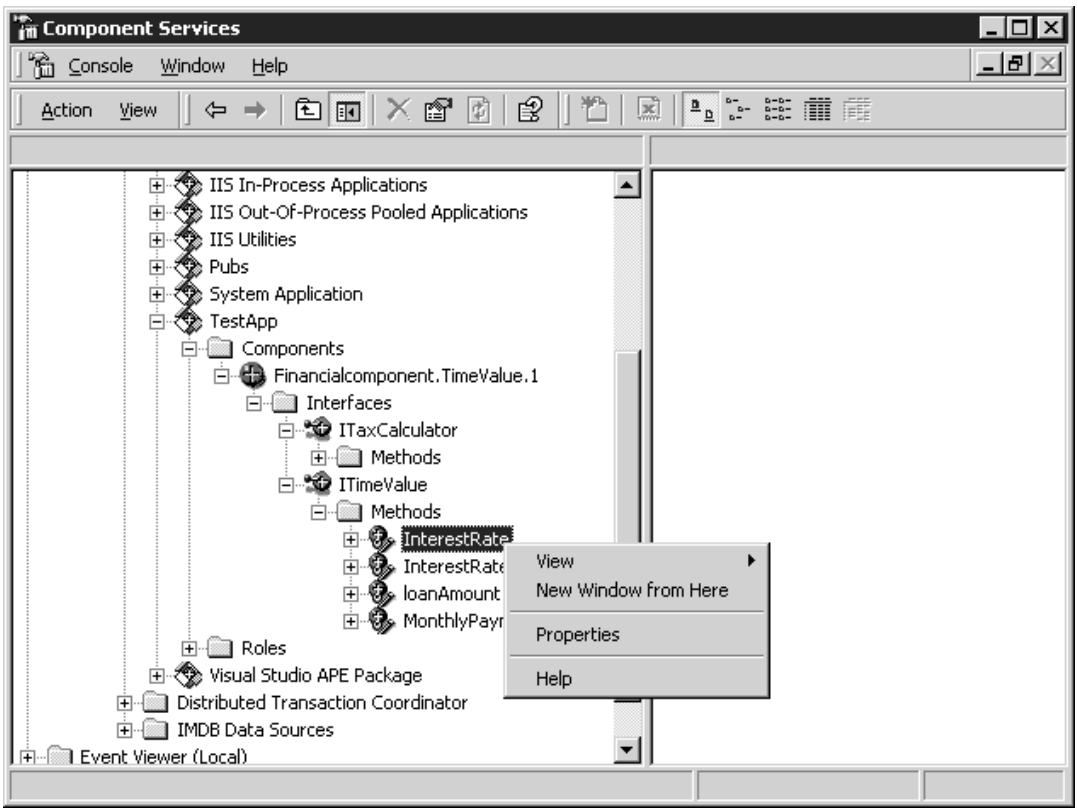



**FIGURE 12.16** Setting the Attributes for a Method.

2. The Method Properties dialog appears as shown in Figure 12.17.
3. Set the `Automatically deactivate this object when the method returns` check box.
4. Click `OK`.

After you set the auto-done bit, then the `Deactivate-on-return` bit is set to true as soon as you enter the method. The state of the transaction vote bit is set based on the return value from the function. If you return `S_OK` from the method, the transaction vote bit is set; this is the same as if you had called `SetComplete`. If the method returns unsuccessfully, for example, it returns an HRESULT indicating an error, then the transaction vote bit is cleared. This is the same as calling `SetAbort`. The user can override the automatic setting of the `Deactivate on return` bit by explicitly calling methods on the `IContextState` interface.

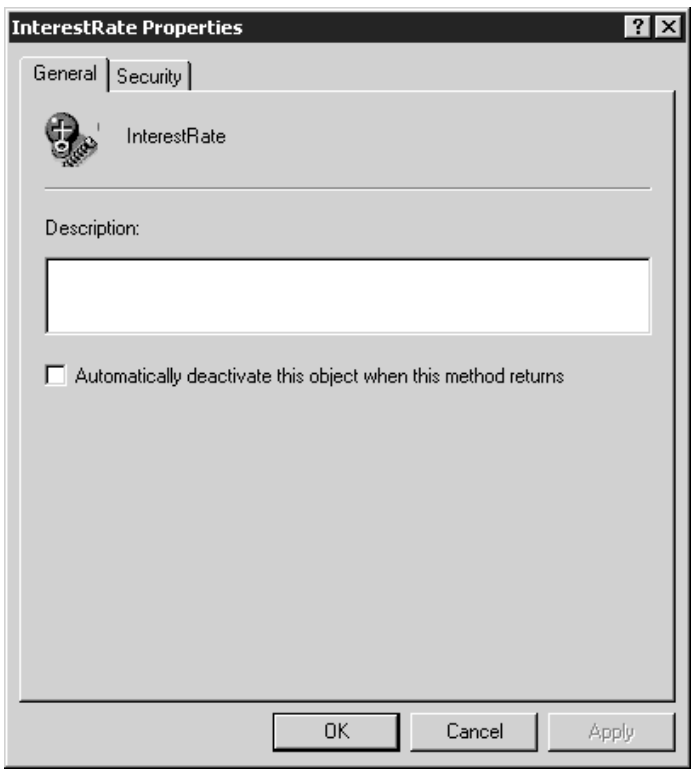



**FIGURE 12.17** The Method Properties Dialog.

## Scalability

COM+ also provides you with a number of services you can use to improve the scalability of your COM objects. In this context, scalability means how your application handles an increased workload.

Imagine if you installed a COM+ component on a server machine and suddenly a thousand people decided they wanted to simultaneously use your server. Each of those clients resides on a separate machine and each of those client applications (as most applications tend to do) creates a COM+ object and then holds on to an interface pointer on that object for the life of the user's session. The problem is that each object uses some quantum of server resources: each object uses some memory and may have its own thread; if your COM+ object does talk to a database, it will also have a database connection. An application designed this way does not scale well. The server machine would likely crash under the load before you reach a thousand users. So how can COM+ improve the scalability of this application?

To understand how COM+ works, imagine you are the manager of a successful rental car company. You know that in a typical year you will have 10,000 customers. Would you run out and buy 10,000 cars to rent to each of these customers? Of course not. Each customer's car would be gathering dust on your lot when that customer was not renting it from you.

You could probably service all of your customers with a pool of 200 cars. When someone arrives and wants to rent a car, you give them the first available car of their requested type. They can use this car for the length of their stay. When they're done, they bring the car back, you clean it up, and then it goes back into your pool to wait for the next customer who wants a car of that type. Maintaining a pool of some scarce resource allows you to service a given number of users of that resource with the smallest possible number of that resource. With COM+, you can achieve the same optimum sharing of a scarce resource using a combination of Just In Time (JIT) Activation and Object Pooling.

### *JIT Activation*

People often write COM client applications so that they instantiate all the objects they need when they are started and then hold on to those objects for the life of the application. Most of the clients that you have built in this book work this way. In many situations, it makes sense to do this because the cost of instantiating a COM object can be high. At a minimum, a registry lookup is required. If the server is out-of-process, then a server process must be started. And if the server is a remote server, then activating an object requires a network round-trip. While it is simpler and seems more efficient from the client's perspective to hold onto object references for the length of a user's session, from the server's perspective this ruins scalability.

A thousand clients may be holding onto their object references but it's likely that only a small number of those clients are actually executing a method on one of those objects at any given moment. Meanwhile, each of those thousand objects is using some quantum of server resources. Without COM+, the solution to this problem would be to place the burden on developers of client applications to be smarter in the way they create and use objects. You can require that they instantiate a COM object only when they need one to execute a transaction and then release it as soon as the transaction is committed or rolled back. This places a great deal of additional work on client developers.

The whole idea of COM+ is to take enterprise-class functionality that would be difficult for most developers to implement in code and make it simple to use. So it is with the functionality I just described. Rather than require the client to manage the efficient allocation and deallocation of objects, COM+ implements this functionality for you. It's called Just In Time (JIT) Activation and is controlled using attributes like all the other services of COM+. Using JIT Activation, when the client first creates an object, the COM+ runtime creates the standard proxy and stub with an RPC channel between the object and its client as shown in Figure 12.18.

When the object sets its `Deactivate-on-return` bit, the COM+ runtime deactivates the object. The object is either destroyed or returned to the object pool (depending on the setting of the object pooling attribute). However, the client still holds the proxy for the object, the RPC Channel is in place, and the stub still exists as shown in Figure 12.19. So as far as the client is concerned, the server-side object is still there. When the client calls a method on the object again, the COM+ interceptor, which sits between the RPC channel and the stub, instantiates another instance of the object (or fetchs an instance from the object pool) and attaches it to the stub and the method can execute as normal.

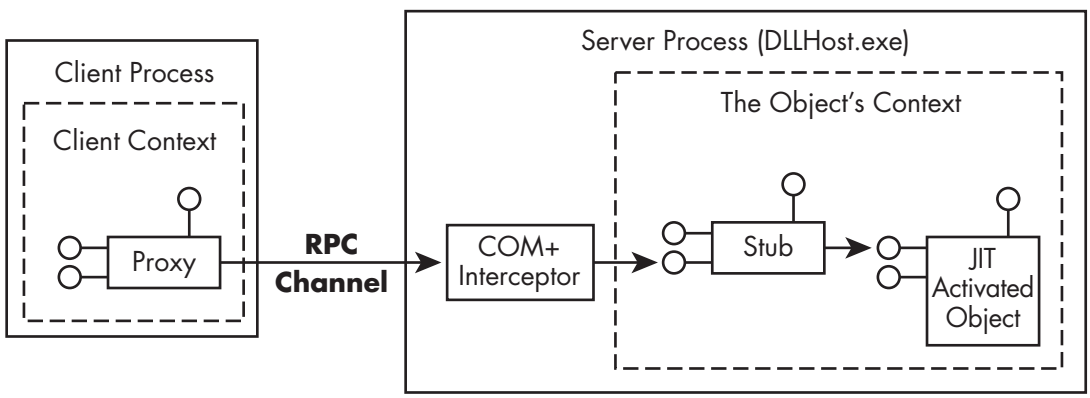



**FIGURE 12.18** An Object That Supports JIT Activation in Its Activated State.

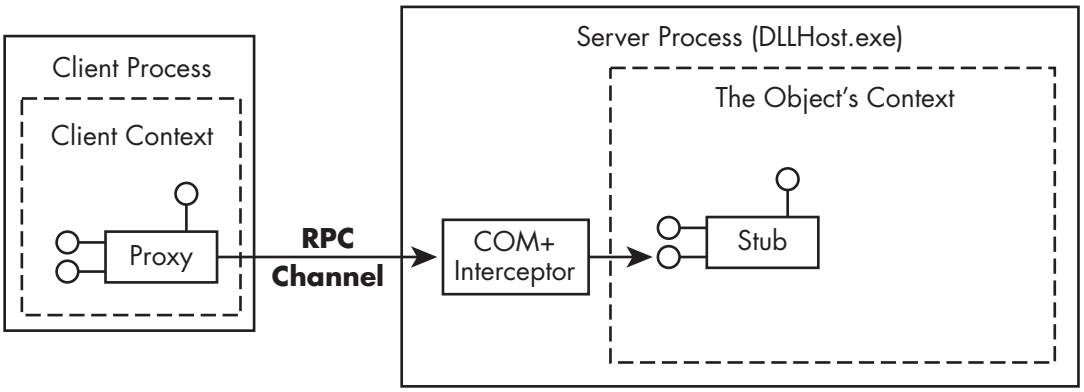



**FIGURE 12.19** An Object That Supports JIT Activation in Its Deactivated State.

All transactional COM+ objects must use JIT Activation; this is to enforce transactional correctness. You can turn on JIT activation on a per-class basis using the Component Services Explorer. Perform the following steps to enable JIT activation for a COM+ component:

1. Right-click on the component in the `Components` folder of your Application.
2. Select `Properties...` from the `Context` Menu (the Component Properties dialog appears as shown in Figure 12.20).
3. Click the `Activation` tab.
4. Set the `Enable Just In Time Activation` checkbox.
5. Click OK.

You can see the dependency between transactions and JIT activation if you first go to a COM+ object in the Component Services Explorer and make it non-transactional by selecting *transactions not supported* for the transaction attribute and then turning off JIT activation. If you then go back to the Transaction tab and try to make it transactional again while JIT activation is still turned off, you see the warning message shown in Figure 12.21 as the system turns JIT activation back on. JIT activation serves two purposes. It not only enforces the correctness of your transactions, it also enhances the scalability of your application by minimizing the number of objects that need to be activated at any given time to service a given number of clients.

### *Object Pooling*

An application that uses JIT activation activates and deactivates a large number of objects. Depending on how the object is implemented, it may be

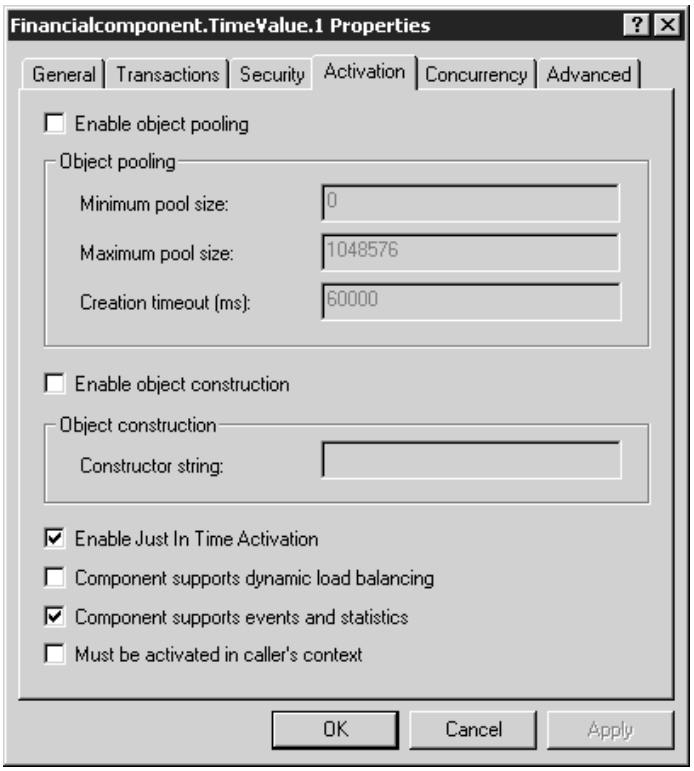



**FIGURE 12.20** The Activation Tab of the Components Properties Dialog.

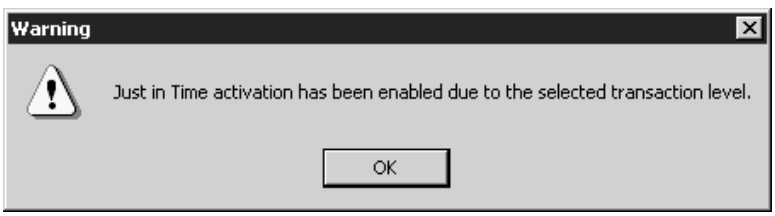



**FIGURE 12.21** Trying to Make an Object That Does Not Support JIT Activation Transactional.

expensive in terms of time to create a new instance of an object whenever you start a new transaction. To alleviate this problem, COM+ supports object pooling. With object pooling, the COM+ pool manager maintains a pool of objects. When a client attempts to activate an object of that type, the COM+ runtime returns an instance from the pool if one is available, instead of creating a new instance from scratch. The client can use the

object for as long as it likes. When the client releases its last reference to the object, or the object returns from a method with its deactivate-on-return bit set, the object is deactivated. If object pooling is turned on, instead of being destroyed, the deactivated object is returned to the object pool as shown in Figure 12.22.

You configure a COM+ class to support object pooling the same way you configure all COM+ services: using the Component Services Explorer. Perform the following steps to *view* the object pooling-related COM+ attributes:

1. Right-click on the component in the `Components` folder of your Application.
2. Select `Properties...` from the `Context` Menu (the Component Properties dialog appears as shown in Figure 12.20).
3. Click the `Activation` tab (shown in Figure 12.20).
4. Set the `Enable object pooling` checkbox.
5. Click `OK`.

Notice I say *view* the pooling-related attributes because most of the COM components that you have created up to now will *not* be poolable. You can tell if an object is poolable if the Enable Object Pooling checkbox

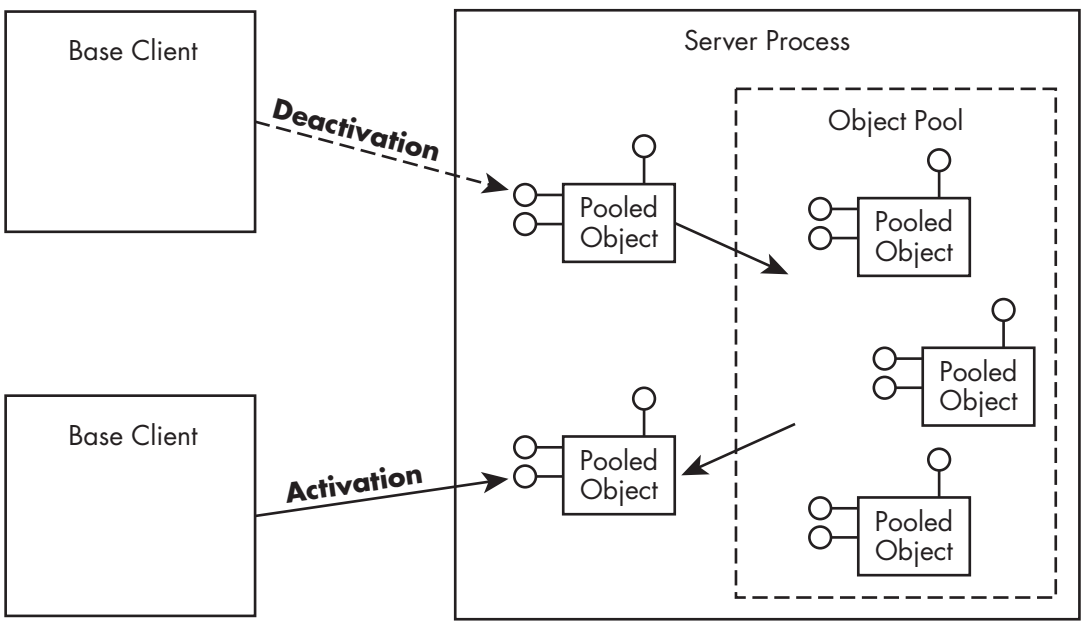



**FIGURE 12.22** Object Pooling.

on the Activation tab of the class-level properties is enabled. A COM+ component must meet the following requirements before it can be poolable.

- It must reside in the Thread Neutral or Free-Threaded Apartment of a process.
- It must be stateless.
- It must not have thread-affinity.
- It must be aggregateable.

Assuming that a COM+ component is poolable, you can configure a minimum number of objects that should be instantiated immediately when the process for the COM+ application starts. You can also specify a maximum number of objects allowed in the pool and a creation timeout. As activation requests are received, the COM+ runtime services them from the initial (minimum) number of objects. If all the objects in this initial pool are being used and additional activation requests arrive, the COM+ runtime creates additional objects to service those requests until the specified maximum number of objects is reached. After that, additional activation requests are queued and they wait the specified creation timeout period for an object to become available before failing. The total number of objects, both activated and pooled (deactivated), is never allowed to exceed the specified maximum.

### *IObjectControl*

There is a problem with object pooling as you understand it so far and it relates to state again. When a client receives an object from the pool, that object has the same state that it had when the last client that used it returned it to the object pool. In most cases, the object should be reinitialized when it is handed to a new client. Moreover, when a client is done with an object and returns it to the pool, you may want to run a termination routine. This termination routine might close a database connection or free some other shared resource the object is using. There is no point in the object holding on to some shared resource while it is sitting idling in the object pool.

If a poolable object wishes to be notified when it is being activated (fetched from the pool) or deactivated (returned to the pool), it should implement the `IObjectControl` interface. The `IObjectControl` interface also allows the object to specify programmatically if it is poolable. Table 12.5 lists the name and the description of the methods in `IObjectControl`.

**TABLE 12.5** The Methods in the IObjectControl Interface

| Method Name | Description |
|---|---|
| Activate | Called when an object is activated. |
| Deactivate | Called when an object is deactivated. |
| CanBePooled | Allows an object to let COM+ know whether it can be pooled. |

COM+ objects that use object pooling *should* implement `IObjectControl`. If an object implements `IObjectControl`, when the object is deactivated (either released by its client or as a result of JIT activation) the COM+ runtime calls the `Deactivate` method on `IObjectControl`. After that it calls the `CanBePooled` method. If `CanBePooled` returns `TRUE`, the object is placed under the control of the pool manager. If the object is later activated on behalf of another (or the same) client, the COM+ runtime calls the Activate method on the object before the object is reactivated.

If a transactional object holds on to a managed resource like a database connection while it is pooled, it must indicate when the resource cannot be used by returning `FALSE` from `CanBePooled`; when `CanBePooled` returns False, it causes the transaction to abort. Also, if a pooled object holds on to a managed resource (like a database connection) while it is pooled, it should turn off the resource manager's auto-enlistment and manually enlist any resources it holds in transactions. The details of doing this are resource-manager dependent and beyond the scope of this book. Suffice to say, you should think long and hard before you decide to have a pooled object hold on to a managed resource like a database connection while it is deactivated in the object pool. The bold code shown in Listing 12.5 illustrates how to implement `IObjectControl` in a COM class that has been implemented with ATL.

**LISTING 12.5** *IMPLEMENTING IOBJECTCONTROL*

```
1.  class ATL_NO_VTABLE CBook :
2.     public CComObjectRootEx<CComMultiThreadModel>,
3.     public CComCoClass<CBook, &CLSID_Book>,
4.     public IDispatchImpl<IBook, &IID_IBook, &LIBID_PUBSBOSERVERLib>,
5.     public IObjectControl
6.  {
7.  public:
8.     CBook()
9.     { }
10.
11. BEGIN_COM_MAP(CBook)
12.    COM_INTERFACE_ENTRY(IBook)
13.    COM_INTERFACE_ENTRY2(IDispatch, IBook)
14.    COM_INTERFACE_ENTRY(IObjectControl)
15. END_COM_MAP()
16. public:
17. // IBook methods go here …
18.
19. // IObjectPool
20.    STDMETHOD_(BOOL, CanBePooled) ()
```

```
21.    {
22.       return TRUE;
23.    }
24.    STDMETHOD(Activate) ()
25.    {
26.    // Initialization logic goes here
27.       return S_OK;
28.    }
29.    STDMETHOD_(void, Deactivate)()
30.    {
31.    // Termination logic goes here
32.    }
33. };
```

### *Database Connection Pooling*

COM+ also increases the scalability of COM objects by helping them effectively share server-side resources like database connections. A good portion of the COM+ components that you write will make a connection to a database. You can manage this connection within your object in one of two ways: (1) you can open a connection to the database when the object is activated and then close the connection when the object is deactivated, or (2) you can open a connection to the database in each method and close the connection before the method returns.

The first approach creates the same scalability problem that you have when a client tries to hold on to a COM object for the lifetime of a user's session: If you have a thousand users currently trying to use your server, then they will have a thousand database connections open. But it is likely that at any given moment only a small fraction of those connections will be used. Meanwhile your database server has to maintain a thousand connections. If enough connections are open, the server slows down to a point where performance becomes unacceptable. Moreover, maintaining a large number of connections can be expensive from a financial standpoint. Many database servers are licensed on a per-connection basis.

The second approach minimizes the number of connections that are used at any time and, thus, improves scalability, but it has an adverse effect on performance. For most RDBMS servers, making the initial connection to the database is a slow operation.

The natural solution to this problem is to maintain a pool of database connections. When a COM+ object wishes to make a connection to a database, it first checks the pool of available database connections. If a suitable connection to the desired database server is already available, the COM+ object can simply grab the connection from the pool and start working.


A suitable connection is one that has the same connection string (i.e., the same server, user name, password, and other connection properties).

If no suitable connections are found in the connection pool, then the object can make a new connection to the database. When the COM object is done with the connection, instead of closing the connection, it simply returns it to the pool where it is available for another object that might need a similar connection.

Now, if you are a true glutton for punishment you could implement this logic yourself, but you don't have to. Once again COM+ makes this functionality available to you without requiring you to write a line of code. As long as you write your COM+ business objects so they access databases using either Open DataBase Connectivity (ODBC) or OLEDB, then your objects automatically use the connection pooling provided by these data access technologies.

### ODBC and OLEDB: Universal Data Access APIs

ODBC and OLEDB are universal data-access APIs. ODBC (which is much older) is implemented as a C language API; OLEDB is implemented as a set of COM interfaces that abstract the functionality of a DataBase Management System (DBMS). The idea is that you write your applications to the ODBC API or use the OLEDB interfaces. Then, to make your application work with any DBMS, you simply get an ODBC driver or an OLEDB provider for the DBMS. An ODBC driver and an OLEDB provider do the same thing: they map calls on the standard interface (ODBC or OLEDB) to calls on the native API of the DBMS. The only difference is the way they are implemented. An ODBC driver is a DLL that implements a specified set of functions. An OLEDB provider is an in-process COM server.

With ODBC, a piece of software called the Driver Manager sits between your application and each ODBC driver as shown in Figure 12.23.

With OLEDB there is no equivalent to the ODBC Driver Manager. Your application talks directly to the OLEDB provider, which talks to your database as shown in Figure 12.24.


The OLEDB Architecture is actually far more complex then that shown in Figure 12.24. I discuss this technology in greater detail in the next chapter. The architecture shown in Figure 12.24 is of sufficient detail for the discussion at hand.

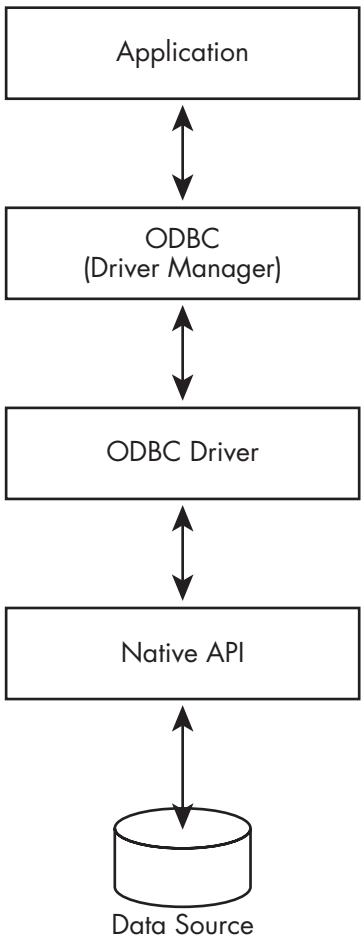
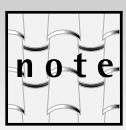



**FIGURE 12.23**
The ODBC Architecture.

**RESOURCE DISPENSERS** • COM+ can implement connection pooling using either ODBC or OLEDB because the ODBC Driver Manager and most OLEDB providers are *Resource Dispensers*.




Check the documentation for your OLEDB provider to see if it implements connection pooling.

In the language of transaction processing, a resource dispenser is a piece of software that works with a transaction manager to manage a pool of some non-durable, shared resource. In this case, the *non-durable*, shared

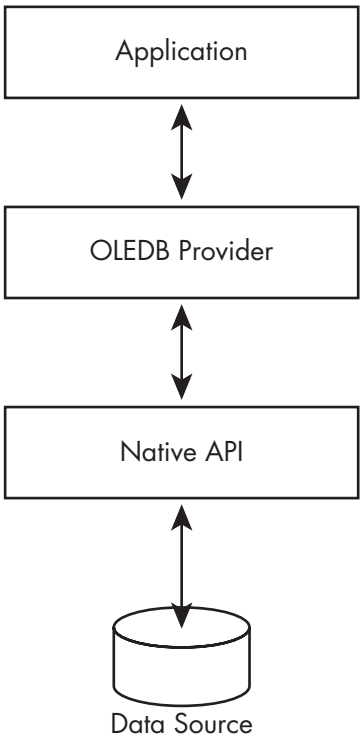



**FIGURE 12.24**

The (Simplified) OLEDB Architecture.

resource is a database connection; it could also be a socket connection or a file. Microsoft publishes an API you can use to create your own resource dispensers.


**A Resource Manager, which you learned about earlier, manages *durable* state and can participate with a transaction manager and other resource managers in implementing the 2-phase commit protocol.**

When you make a connection to a database server using OLEDB or ODBC, the ODBC Driver Manager (in the case of ODBC) and some OLEDB providers, first check their database connection pool to see if a connection to the requested database server already exists with the same connection string. The connection string includes the user ID and password and sometimes other parameters. If an unused connection exists in the pool that has a matching connection string, the Driver Manager or Provider returns the already open connection. The object can now use the connection as it sees fit. When the object is done with the application, it closes the connection.

Instead of destroying the connection, the resource dispenser returns the connection to the pool where it can be reused again.

To get the most out of connection pooling (or any other pooled resource that is managed by a resource dispenser), you need to follow two simple programming tenets: (1) acquire the resource as late as possible, and (2) release the resource as soon as possible. Remember the whole idea of resource pooling is that you have a resource that is in short supply; you don't want to hold on to the resource if you are not using it. So, the methods in your COM+ objects should create a connection only in a method that actually executes operations on the database. The method should close the connection before it returns. If you've done database programming before, you know that without connection pooling, using an approach like this can have poor performance. Creating a connection on a remote database is usually an expensive operation in terms of time. But with connection pooling, this is not a problem.

You do not configure database connection pooling in the Component Services Explorer. Database connection pooling is not specific to COM+. Any application using ODBC can take advantage of connection pooling. Perform the following steps to examine the ODBC connection pooling parameters in effect on your machine:

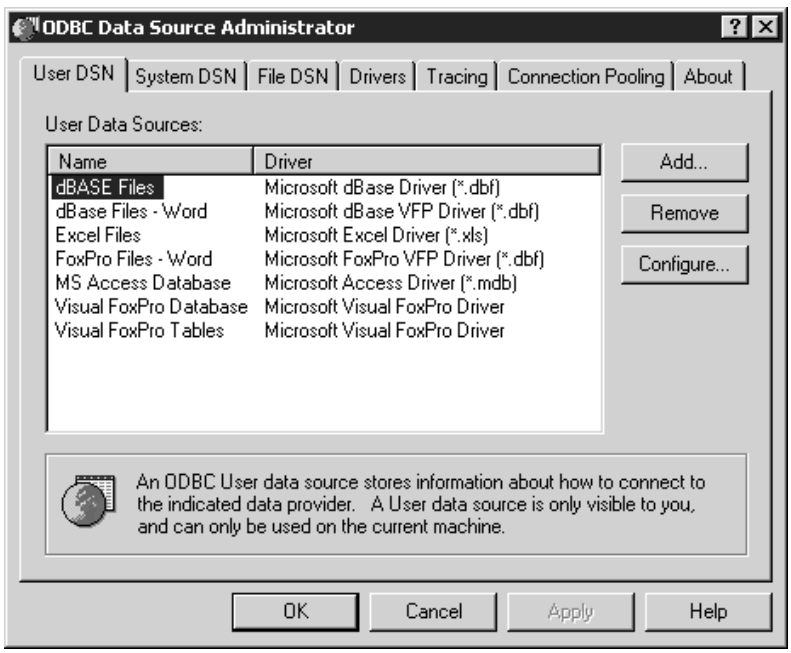



**FIGURE 12.25** The ODBC Data Source Administrator.

1. Click the `Start` button.
2. Select `Programs\Administrative Tools\Data Sources` (the ODBC Data Source Administrator appears as shown in Figure 12.25).
3. Click the `Connection Pooling` tab (you should see the window shown in Figure 12.26).
4. Double-click the SQL Server entry in the ODBC Drivers list (the Set Connection Pooling Attributes dialog appears as shown in Figure 12.27).

Notice that in Figure 12.27 you can configure whether the driver pools the connections and how long an open connection remains in the pool.

If you are using OLEDB, you have to check the documentation for your OLEDB provider to see if it supports connection pooling and how to configure connection pooling for the provider.

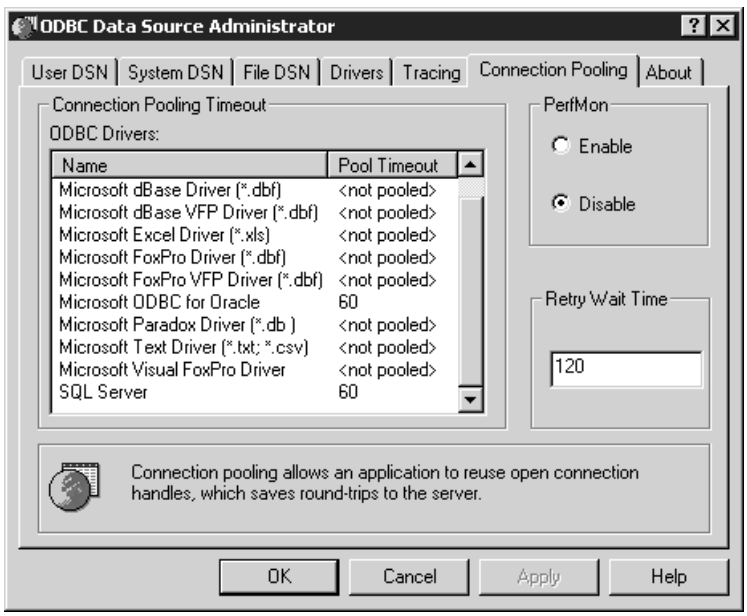



**FIGURE 12.26** The Connection Pooling Tab of the ODBC Data Source Administrator.

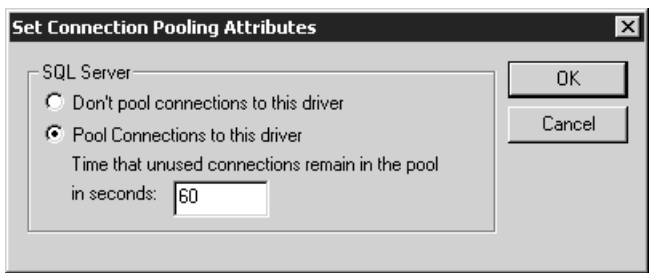



**FIGURE 12.27**

The Connection Pooling Attributes Dialog.

## Synchronization

Both single and multi-threaded clients may use COM+ objects. You want your objects to be usable by both types of clients but you don't want to require developers of COM+ objects to have to write complicated thread synchronization code for all of their objects. COM used Apartments to provide thread-safety for objects that were not coded to be thread-safe. If your object is thread-safe, you can specify that the object should run in the Multi-Threaded Apartment (MTA). In this case, multiple threads may access the object simultaneously. MTA objects have the best performance when used in a multi-threaded environment, but the developer of this object has to implement code to make the object thread-safe. If an object is not thread-safe, you can specify that it should only run in the Single-Threaded Apartment (STA). An STA is intimately tied to one thread and only that thread is allowed to directly call a method on a COM object in an STA.

Unfortunately, STAs are not a good thread-safety solution for COM+ objects. The main problem is that STA objects also have thread-affinity. To maximize concurrency and performance, the surrogate process that COM+ uses is multi-threaded. You would like this process to have the liberty to choose which thread it uses to best service a particular method call. This runs counter to the whole concept of COM apartments. Apartments place strict limits on *which* thread is allowed to call a COM object directly. If the COM+ runtime tried to call into the object using a thread that is not the one thread associated with the object's STA, then intra-process marshaling must be used to send the method call to the thread in the object's STA. This thread switch is almost as slow as cross-process marshaling.

In addition, STA objects are unsuitable for use in an object pool. If it is not clear why this is so, imagine that you did try to implement object pooling with STA objects. Let's say you have a pool of five STA threads (Threads 1-5) that are to be used to create and manage a pool of STA COM objects. Thread 1 creates object A, executes some methods on the object, and then instead of deactivating the object when it is done with it, it places the object in the object pool. It's work done, thread 1 goes back to the thread pool. Now imagine that another activation request is received for a different type of object, and thread 1 is used to service this request. While thread 1 is busy servicing this request, an activation request for an object of the same type as object A is received. Because object A resides in the pool, it's natural that this object should be used instead of creating a new object (that's what object pooling is all about). But you have a problem. Because object A is an STA object, it can only be used by thread 1, which is busy servicing another request. So, even though there are four other threads available, any of which could service the request, you have to wait until thread 1 is free before you can use object A again.

As I mentioned in Chapter 9, the solution to all of these problems is the new COM+ Thread-Neutral Apartment (TNA). A thread in any apartment in a process can access an object that lives in the TNA. This allows the COM+ runtime to have the highest performance and concurrency that it possibly can and it also allows the COM+ runtime to implement object pooling. The problem here is that if you allow any thread to call into an object anytime it wants, you are faced with the very problem that apartments were designed to solve: How do you make your COM+ objects thread-safe without writing a lot of complex code?

Once again, COM+ and Attribute-Based Programming is the answer. COM+ provides activity-based synchronization support. You can specify that you want the system to provide thread synchronization support for your COM+ objects using the synchronization attribute.

Providing thread-safety for COM+ objects is more complex than just keeping a per-object lock and allowing only one thread to execute an object's methods. A COM+ object may call the methods of another COM+ object that resides in a different process, maybe even on a different machine. You would not want to get into a situation where one object is adding items to an order as another item is canceling the order. Using COM+'s synchronization support, you can synchronize the processing of multiple objects so no unwanted concurrency occurs even if the objects reside on different machines. To implement this functionality, COM+ introduces the concept of an *activity*.

### *Activities*

An activity is defined as a set of objects performing work on behalf of a single client, within which concurrent calls from multiple threads are not allowed. Think of an activity as a logical thread that can span processes and machines. COM+ makes sure that two physical threads are not running at the same time within this logical thread of activity. This is actually a much stronger way of saying that more than one thread is not allowed to call into an object at the same time. You can specify that an object should support synchronization using the Component Services Explorer. There are five possible settings for the synchronization attribute: Table 12.6 gives the name and a description of each of these settings.

**TABLE 12.6** Possible Settings for the Synchronization Attribute

| Synchronization Setting | Description |
|---|---|
| Not Supported | The Object does not run within an activity. |
| Supported | The object runs within an activity if its creator runs within an activity. |
| Required | The object runs within its creators activity if the creator had one. It creates its own activity if its creator does not have one. |
| Requires New | The object always creates its own activity. |
| Disabled | This object ignores completely the value of the synchronization attribute when determining context placement. This setting makes your configured component behave like an unconfigured one. |

Perform the following steps in the Component Services Explorer to set the Synchronization attribute for a COM+ component:

1. Right-click on the component in the `Components` folder beneath the Application.
2. Select `Properties` from the `Context` menu (the Component properties dialog appears).
3. Click the `Concurrency` tab (see Figure 12.28).
4. Set the radio button for the attribute that you would like to set.
5. Click `OK`.

It's likely that most of the options you see in this Window are disabled. There are a couple of important dependencies regarding synchronization that effect which settings are available.

- All STA components must reside in an activity whether or not they are transactional. An STA component, therefore, is only allowed to have either the Required or Requires New settings.
- All transactional COM+ components must reside within an activity so they also may only choose between the Required and Requires New settings.
- An object that supports JIT Activation whether it's transactional or not is allowed only the Required and Requires New settings.

An object that is running with COM+ synchronization support has an Activity ID in its Context just like an object that is running within a transaction has a Transaction ID in its Context. You can retrieve the Activity ID for an object by calling the `GetActivityId` method on the `IObjectContextInfo` interface of your object context. Let's see how activities work by looking at a single, synchronized object (in this case, let's consider an object whose synchronization attribute is Required and its creator is not currently running within an activity).

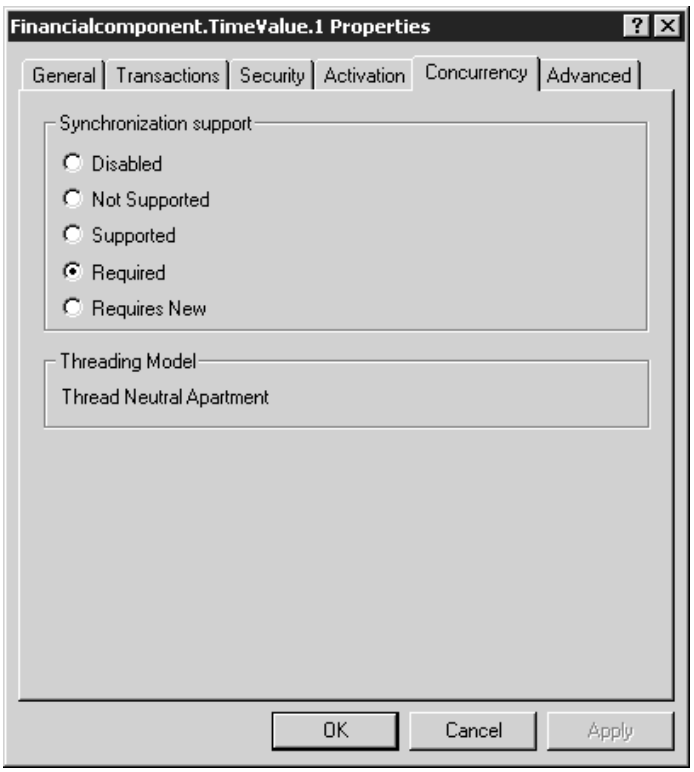



**FIGURE 12.28** Configuring Synchronization for a COM+ Component.

When the object is first created, a new activity is created. Each activity contains a process-wide synchronization lock. Whenever a thread calls a method on the object, it acquires this lock. The lock is not released until the thread returns from the method. If a second thread tries to call into the method, it also attempts to lock the synchronization lock for the object's activity. This second thread cannot obtain this lock and enter the method until the first thread returns. Now let's consider what happens if, while within a method, a thread attempts to activate a second COM+ object.

Whether the activity flows to this second object depends on the synchronization setting that this second object has. Table 12.7 gives the activity status for an object when created from an object that is currently running within an activity.

**TABLE 12.7** Activity Status of an Object Whose Creator Is within an Activity

| Synchronization Setting | Activity Status |
|---|---|
| Not Supported | Does not run within an activity. |
| Supported | Runs within its creator's activity. |
| Required | Runs within its creator's activity. |
| Requires New | Runs within its own activity. |

If you understood how transactions are created and propagated from object to object, then understanding how activities propagate from object to object isn't a big step for you. Let's say this second object has a synchronization attribute of Required. It shares the activity and, hence, the process-wide synchronization lock with its creating object. For a thread to make a method call on this second object, it must obtain this process-wide synchronization lock. If a method in the first object is still executing, it cannot do this. So a method call on the first object and a method call on the second object are never running simultaneously. If this second object has a synchronization setting of Not Supported or Requires New, it does not run within the same activity as its creating object so a method call on this second object may run concurrently with a method call on the first object.

Let's think about what happens if the first object activates an object that resides on a separate machine. The synchronization attribute still flows from the first object to the newly activated second object. If the second object is marked with a synchronization attribute of Supported or Required, then this second object shares the same activity as its creator. Because these objects are in different processes (on different machines), they cannot share the same physical thread. Not to worry, the COM+ runtime on the second machine creates a process-wide lock on the other machine.

There still is another problem, however. Consider what happens if the first object makes a method call to the second object and this second object makes a call back to the first object. The first object is blocked waiting for its method call on the second object to return (remember, method calls in COM and COM+ are synchronous by default). So the second object also blocks when it makes its call back to the first object. This is the classic case of deadlock. COM+ handles this situation using the concept of *Causality* (actually DCOM does the work here). Causality is a logical name for a set of nested function calls. A causality ID (which is a GUID) is sent along with every method call made via DCOM. When the second object calls back to the first object, it uses the same causality ID that the first object used to initiate the sequence of method calls. The first object checks the causality ID and sees that it is the same as the causality ID that it is waiting on. The first object realizes that a deadlock will be caused if it does not allow the call through, so it allows the method call to continue.

## Queued Components

Think for a moment about your telephone. If you ignore answering machines and voice mail for a moment, when you call someone on the telephone, they must be available to answer the phone or no communication can take place. This is an example of synchronous communication. Both the sender and the receiver must be ready to receive the communication and their lifetimes must overlap. But waiting until someone is in his or her office and ready to receive a phone call is inconvenient. In many cases, it's not important that you talk to the person at that exact moment, you just need to get a message to them that they can listen to whenever they are ready. That's why voice mail was invented. With most telephones used in a business setting, if the person is not available to receive the phone call, you are automatically given the person's voice mail. You leave your message and the person will receive your message at a later time. This is an example of asynchronous communication. E-mail is another example of asynchronous communication. When you send an e-mail to someone you wouldn't expect your e-mail to be rejected just because the person you are sending the e-mail to is not available to receive it at that exact moment. You expect the message to be sent to the e-mail server on that person's domain where it patiently sits until the person is ready to read it. The COM+ Queued Components (QC) Service allows you to implement the equivalent of Voice Mail for your COM+ components. When using COM+ objects without the Queued Components Service, the client and object communicate over an RPC channel using a marshaling proxy and stub as shown in Figure 12.29.

The client has a proxy in its address space that turns a method call on a COM interface into an RPC message that is sent to a stub on the server machine. The stub receives and interprets the message and sends return values and output parameters back to the proxy. If someone accidentally killed

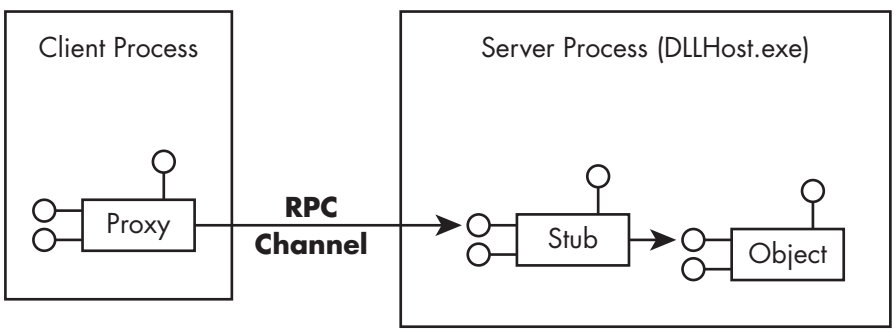



**FIGURE 12.29** A COM/COM+ Client and Server Component without the Queued Components Service.

the server process or if for any other reason the server was not available at the time you made the method call, everything would fail. With Queued Components, the situation changes completely as in Figure 12.30 illustrates.

As shown in Figure 12.30, once you configure a COM+ class to use the QC Service (you'll see how to do this shortly), the COM+ runtime creates a set of queues using Microsoft Message Queue (MSMQ) between the client and server. When an object of a QC class is instantiated, the COM+ runtime creates a QC Recorder on the object's client. When you make a method call on this queued component, the QC recorder on the client transparently turns the set of method calls that you make into a single MSMQ message. When the object is deactivated, the QC service sends this message to the queue that the COM+ runtime had setup when you configured the component to support Queued Components.

On the server side, a component called the QC Listener Helper checks the queues looking for messages as shown in Figure 12.30. When the QC Listener Helper sees a message waiting in the queue for a component, it extracts the message from the queue and forwards it to a system-provided component called the QC Player. The QC Player then plays the method calls back to the component. Because the system is using MSMQ as its transport mechanism, you receive all the robustness and availability benefits that MSMQ provides. MSMQ caches the messages in its persistent store. If the server that the component is running on is down at the time the message is sent, this is not a problem. The next time the server comes up the QC Listener Helper reads the Queue, sees there is a message, and forwards the message to the QC Player that plays the method calls back to the queued object. As I said, it's conceptually similar to Voice Mail. In many ways, Queued Components is just a simplified way of using MSMQ.

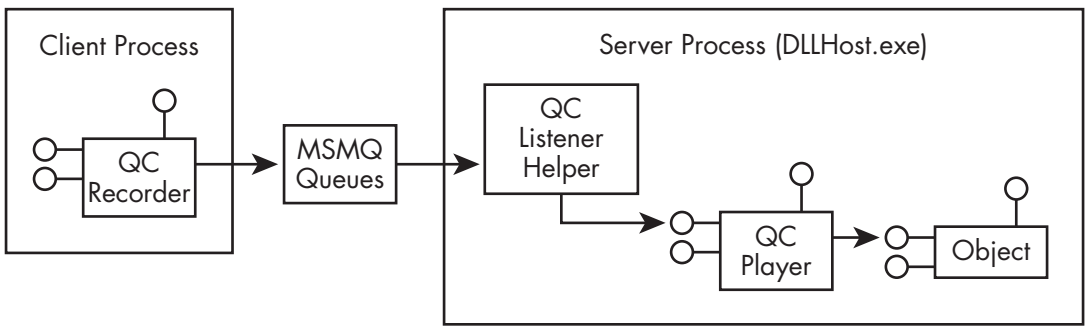



**FIGURE 12.30** A COM+ Client and Server Component Using the Queued Components Service.

### *Design Restrictions on Queued Components*

You have to be careful how you design your components if they are to function successfully as Queued Components. To run as a Queued Component, a COM component must obey the following rules:

- The interfaces the component implements cannot contain output parameters.
- The component cannot return application-specific HRESULTS.
- The methods implemented by the component cannot depend on additional information or interaction from the client after the initial method call.

#### NO OUTPUT PARAMETERS

Queued Components cannot implement interfaces that contain output parameters. These are parameters defined with the IDL attributes `[out]`, `[in,out]`, or `[out,retval]`. So the `ISpellChecker` interface, which was defined as shown below, could not be implemented as a Queued Component because the parameter `isCorrect` on the `CheckSpelling` method is an output parameter:

```
interface ISpellChecker : IUnknown
{
   HRESULT CheckSpelling([in,string] char *word,[out,retval] BOOL *isCorrect);
   HRESULT UseCustomDictionary([in,string] char *filename);
}
```

Queued Components cannot implement output parameters because the client will most likely not be around to receive the return values. When the client invokes a method on a Queued Component, it has no idea when that method will be executed. It is quite possible that the client itself will have terminated long before the queued component ever executes the method call. In fact, all of the rules previously stated basically arise from a single rule you must keep in mind when writing Queued Components: Never assume that the lifetime of the client and server will overlap.

#### NO APPLICATION-SPECIFIC HRESULTS

If you understand the basic rule about no overlapping lifetimes, then it should be clear why a queued component couldn't return an application-specific HRESULT. Who will be there to receive the HRESULT? When a client calls a queued component, the COM+ runtime returns an HRESULT that tells whether the message was successfully queued. After that, the client has no idea when the server actually executes the method.

#### NO DEPENDENCIES ON ADDITIONAL INFORMATION

Once you call a method on a Queued Component, the method that is called cannot depend on receiving additional information from the client. So the Stock Server component that you built in Chapter 8 definitely won't work as a Queued Component. This component cached an interface pointer that was implemented by the client, and the server invoked a method on this cached interface pointer whenever a stock price changed by more than a certain delta. This component would not work as a Queued Component because by the time the component runs, there may not be a client available for the server to call back on. When you get right down to it, the real client of the Queued Component is the QC Player that plays back method calls sent via MSMQ from the QC recorder on the client. The QC player knows nothing about the stock server's callback interfaces.

#### WHAT IF YOU MUST RECEIVE A RETURN VALUE?

What if you want to use the Queued Component Service, but your client application must be able to receive a return value from its queued components? A client application can receive a response from a queued component using a *response object*. A response object must be configured as a queued component itself. The client must instantiate this response object specifying the client machine as the destination server for the object. The client then passes the response object as one of the input parameters to the queued component (you must modify your queued interfaces so they accept this additional parameter). The queued component can then call methods on the response object when it wants to send its return values back to the client. The queued component actually sends its return values back to the response object not to the client application. That's why you must specify the client machine as the destination server for the response object. You want the response to return to the client machine. The client application does not have to be running when the queued component calls the response object. Typically, a response object writes to a database, sends an e-mail, or writes to a log file. The Queued Component example that you'll build in Chapter 14 won't use a response object, but the Web site for this book contains an enhanced version of this example that does use a response object.

### *Queued Components and Transactions*

One of the best features of COM+ Queued Components is that they work with COM+ transactions. The underlying transport mechanism for Queued Components, MSMQ, is a COM+ Resource Manager just like SQL Server or Oracle (see the sidebar entitled the Role of MSMQ in COM+). If you send a message to an MSMQ queue within a transaction, MSMQ buffers the message until the outcome of the transaction is known. If the transaction com-

mits, the message will be sent to the queue; if the transaction aborts, the message is not sent to the destination queue. Queued components can piggy back on this mechanism to allow you to make transactional method calls. The QC recorder takes on the same transaction attribute as its client. If the client is currently running within a transaction, the QC recorder enlists in the transaction. When the QC recorder is deactivated, it sends its message to MSMQ within the context of this transaction. If the transaction is aborted, the message is not sent to the queue, so it's as if the queued method calls never happened. Queued components also use transactions on the server side. The QC Listener Helper is a transactional component. It dequeues the message and sends it to the QC player within the context of a transaction. If the method call fails, the QC Recorder aborts the transaction and the message is returned to the queue so it can be tried again.

### *Configuring a Queued Component*

As always, you configure Queued Components using the Component Services Explorer. To use the COM+ Queued Component service, first make your COM+ Application Queued. This causes COM+ to create the necessary message queues for your application. To make your COM+ application queued, perform the following steps:

1. Right-click on the COM+ application in the Component Services Explorer.
2. Select `Properties` from the `Context` Menu (the Application Properties dialog appears as shown in Figure 12.31).
3. Click the `Queuing` tab (shown in Figure 12.31).
4. Set the `Queued` and `Listen` checkboxes (the Listen checkbox enables when you set the Queued checkbox).
5. Click `OK`.

These steps cause COM+ to generate the message queues needed for your application. You can view these queues by going to the Computer Management Explorer, which is a Snap-In for the Microsoft Management Console just like the Component Services Explorer. To view the main queue for the application, perform the following steps in Windows 2000:

1. Click the `Start` menu.
2. Select `Programs\Administrative Tools\Computer Management` (the Computer Management Explorer appears as shown in Figure 12.32).
3. Open the `Services and Applications` item in the Tree on the left side of the Windows.
4. Open the `Message Queuing` item in the tree.
5. Open the `Public Queues` folder.

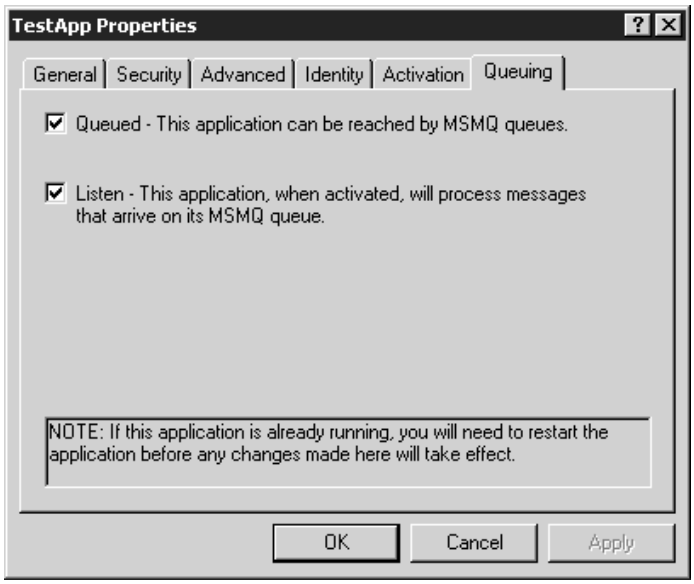



**FIGURE 12.31** Configuring a COM+ Application to Support Queuing.

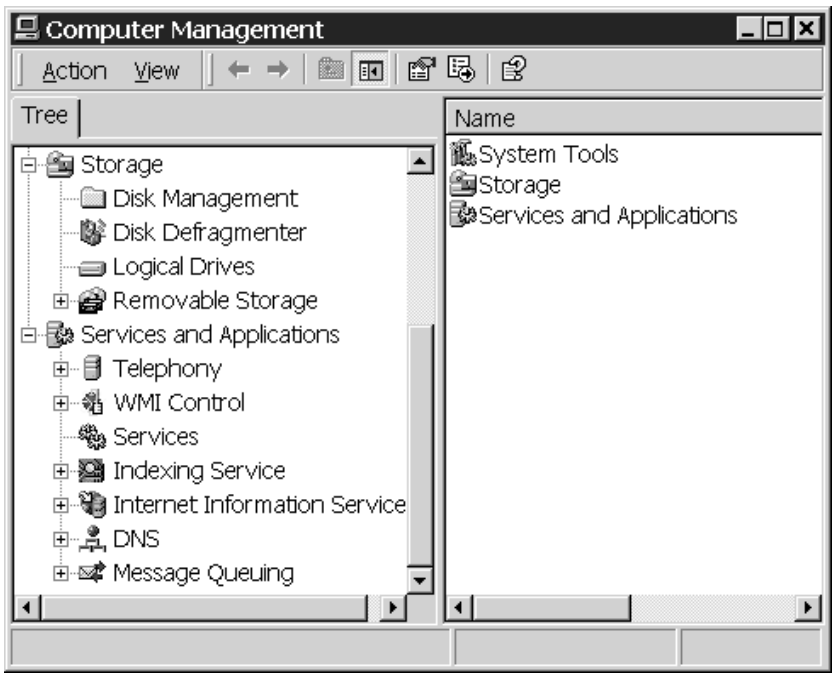



**FIGURE 12.32** The Computer Management Explorer.

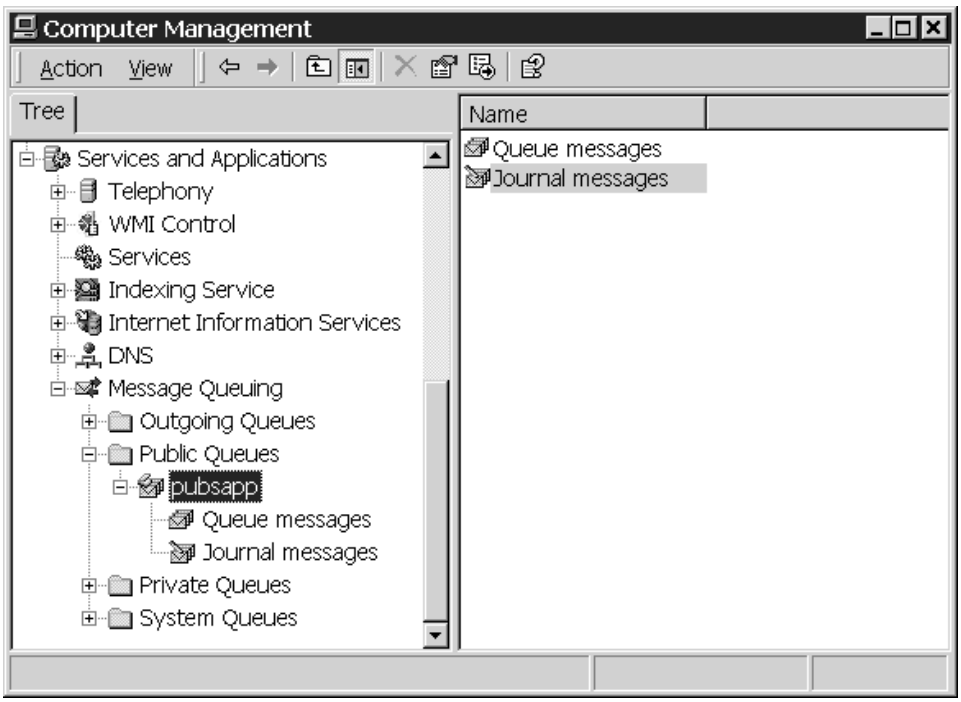



**FIGURE 12.33** The Main Queue for a COM+ Application.

You should see a queue beneath there that has the same name as your COM+ application as shown in Figure 12.33.

The queue shown in Figure 12.33 is the queue for a COM+ Application called pubsapp. If the calls that you make to a queued interface in the pubsapp COM+ application succeed the first time, this is the only queue that is used. COM+ also creates additional queues that are used if the Queued Component Service tries and fails to deliver a message to a queued component. I talk about these other queues shortly when I discuss poison messages.

After you make a COM+ application queued, you must specify which interfaces within the COM+ Application should be queued. Perform the following steps to make an interface queued:

1. Right-click on an interface in the Component Services Explorer as shown in Figure 12.34.
2. Select `Properties` from the `Context` menu (the interface properties dialog appears as shown in Figure 12.35).
3. Click the `Queuing` tab (shown in Figure 12.35).
4. Set the `Queued` checkbox.
5. Click `OK`.

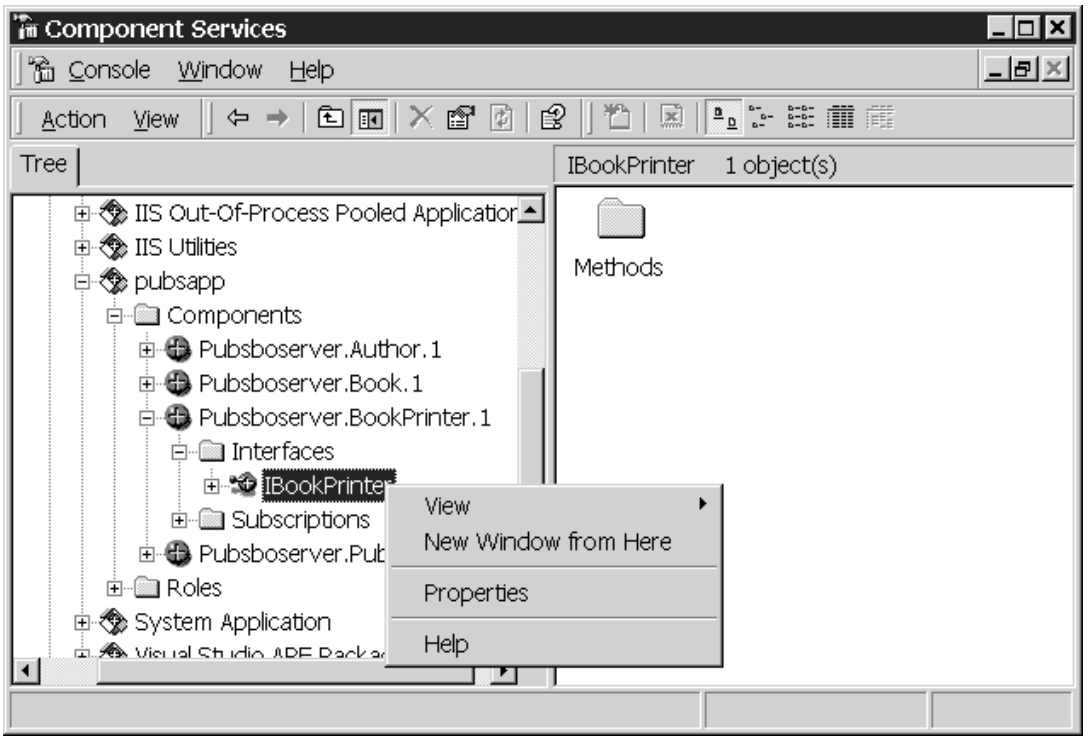



**FIGURE 12.34** The First Step to Making an Interface Queued.

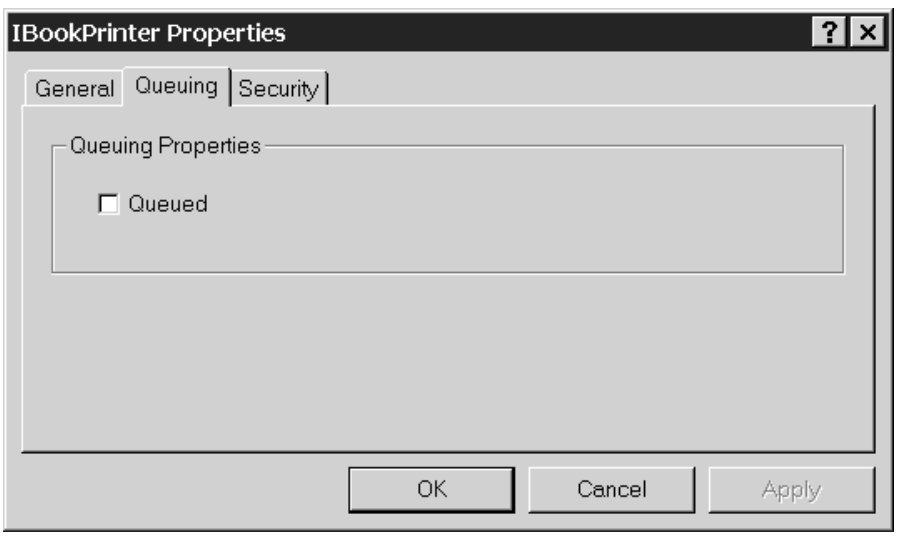



**FIGURE 12.35** Enabling Queuing for an Interface in the Application.

If the Queued checkbox is disabled, that means the interface cannot be queued. Most likely this is because the interface violates one of the design rules for queued components.

### *Using Queued Components*

Once you have configured your components to be queued, there are a few things that you have to do to use them via the queued component service. First, you must instantiate your component using the Queued Moniker.


In the introduction to this book, I mentioned that it was my intention to concentrate on the 90% of COM that application developers most often use. I agonized over whether to include Monikers. Eventually, time constraints made the decision for me and I could not create an entire chapter on Monikers. Since Monikers are used in Queued Components, I decided to cover them (albeit briefly) in this chapter. If you already understand Monikers, you can skip ahead to the section entitled *The Queue Moniker*; if not, go ahead and read this section on Monikers. It is not critical that you understand all the information in this section (especially on your first read through). If you find yourself getting lost, by all means skip ahead.

### *What Is a Moniker?*

So far, you have only created COM objects using the `CoCreateInstance(Ex)` functions. In some scenarios, these functions are not sufficient. For instance, what if you want to store the state of object persistently in a file and then reconstitute the object from the information in that file. What if you just want to assign a textual name to an object that is being persisted to a database or file so you can reconstitute it later using this name. Or what if you want to do something even more complex than that. Perhaps you have a COM object located at an Internet URL and when the user attempts to instantiate the object, you would like to download the object's server to the local machine, register it, and then instantiate the object in the normal manner (this is how ActiveX controls are used over the Internet). The solution to all of these problems is *Monikers*.

A `Moniker` is a COM component that implements the `IMoniker` interface. Think of a `Moniker` as an object that lets you locate and activate other objects. The business logic associated with locating and activating an object is unique to each particular type of `Moniker`. There are a number of standard `Monikers` that are provided for you by COM and you can also create your own.

You call the `BindToObject` method on the `IMoniker` interface to create a COM object from a `Moniker`; this is called binding the `Moniker`. For instance, the File `Moniker` (one of the standard `Monikers` provided by the COM) lets you create an object from the contents of a file. To use the File Moniker, you first call the `CreateFileMoniker` API function to create a File Moniker. `CreateFileMoniker` is defined as follows:

```
HRESULT CreateFileMoniker(
   LPCOLESTR pPathName, // [Input] The path of a file
   IMoniker  **ppMoniker // [Output] The IMoniker interface pointer.
   )
```

Next, you call the `BindToObject` method to bind the contents of the file to an object. The logic within the File Moniker first finds the CLSID of the COM class associated with the file contents. The File Moniker creates an instance of this class and passes the contents of the file to the object. The File Moniker has a pre-defined search strategy it uses to find the CLSID of the COM class associated with the file contents. It first searches the contents of the file for a CLSID. If it doesn't find the CLSID in the file, it looks in the registry for the server object associated with the file extension (for instance, a `.doc` is associated with the Microsoft Word Document object).


**The `CreateFileMoniker` function uses the `GetClassFile` API function to find the CLSID of its associated COM class. See the documentation for `GetClassFile` for an in-depth discussion of how this search strategy works.**

The File Moniker instantiates an instance of the COM class that it finds. It will `QueryInterface` the object for the `IPersistFile` interface and then pass the contents of the file to the `Load` method of `IPersistFile`. The COM class associated with the file contents must implement the `IPersistFile` interface to work with the File Moniker. The code shown in Listing 12.6 creates a File Moniker using a .doc file and then binds the Moniker to an instance of its associated COM class: the Microsoft Word Document.

**LISTING 12.6** *USING THE FILE MONIKER*

```
1.  void CMonikertestclientDlg::OnTestfilemonikerButton()
2.  {
3.     HRESULT hRes;
4.     IBindCtx *pBindCtx;
5.     IMoniker *pMoniker;
6.     IDispatch *pDispatch;
7.     _Application WordApp;
8.     _Document WordDoc;
9.
10.    hRes=CreateFileMoniker(L"c:\\alan\\filemonikertest.doc",&pMoniker);
11.    if (FAILED(hRes))
12.    {
13.       _com_error err(hRes);
```

```
14.       AfxMessageBox(err.ErrorMessage());
15.    }
16.    hRes=CreateBindCtx(0,&pBindCtx);
17.    hRes=pMoniker->BindToObject(pBindCtx,NULL,IID_IDispatch,
          (void **)&pDispatch);
18.    pBindCtx->Release();
19.    pMoniker->Release();
20.    if (FAILED(hRes))
21.    {
22.       _com_error err(hRes);
23.       AfxMessageBox(err.ErrorMessage());
24.    }
25.    WordDoc.AttachDispatch(pDispatch);
26.    WordApp.AttachDispatch(WordDoc.GetApplication());
27.    WordApp.SetVisible(VARIANT_TRUE);  // Microsoft Word Will Appear
28. }
```

On line 10 of Listing 12.6, you create a File Moniker using the `CreateFileMoniker` function. To bind any Moniker to an object, you need a Bind Context, an object that stores information about a particular Moniker's binding operation. On line 16, you call the `CreateBindCtx` API function to create a Bind Context. On line 17, you call the `BindToObject` function on the File Moniker and request the `IDispatch` interface. This returns the default interface on the Microsoft Word Document Object. In this code, you used the MFC COleDispatchDriver to make it simple to work with the Automation interfaces exposed by Microsoft Word. You created two `COleDispatchDriver`-derived classes (`_Document` and `_Application`) using the MFC ClassWizard and the Microsoft Word type library. On line 25, you attach the `IDispatch` pointer you received from the File Moniker to a `_Document` object, then on line 26 you call `GetApplication` on the `Document` object and attach the return value to an `_Application` object. Finally, on line 27, you make Microsoft Word Visible. Notice how the File Moniker hides all the file handling logic involved with opening a Word Document. All of this logic is encapsulated within the File Moniker.

There are several other system-provided Monikers available including the URL Moniker (used to download and activate ActiveX controls given a URL), the Class and New Monikers (used to instantiate new objects given a CLSID), the Item Moniker (that can be used to create an object from a specified portion of a file), and the Composite Moniker (used to combine two or more Monikers). Remember, you can also create your own custom Monikers. I don't want to go into each one of these Moniker types, but I encourage you to read more about them (and other Moniker types) in the MSDN library (`msdn.Microsoft.com/library` or on CD-ROM).


I talk about the New Moniker and the Class Moniker after I discuss the `MkParseDisplayName` function.

Most Monikers have a Create function. The COM API contains a `CreateFileMoniker` function and a `CreateClassMoniker` function. However, the easiest and most flexible way to create a Moniker is using the `MkParseDisplayName` function. `MkParseDisplayName` is defined as shown here:

```
HRESULT MkParseDisplayName(IBindCtx *pbc,
        LPCOLESTR strDisplayName,
        ULONG *pEaten,IMoniker **pMoniker);
```

The `MkParseDisplayName` function creates a Moniker using a user-friendly string called a *Display Name*. The general format of a Display Name is as follows:

```
MonikerType:MonikerSpecificInformation
```

The display name for a File Moniker is constructed as follows:

```
File:Path
```

You can omit the File: prefix if you like. Two (equivalent) Display Names for a File Moniker are shown here:

```
File:c:\alan\filemonikertest.doc
c:\alan\filemonikertest.doc
```

The Display Name for a class Moniker is defined as follows:

```
Clsid:{CLSID}
```

An example Display Name for the class Moniker is shown below. You can omit the curly braces if you want:

```
Clsid:{ CF166FD0-958C-11D3-A0F5-00A0CC330C70}
```

You can construct a display name for the New Moniker using either the CLSID or the ProgID of a COM class. Two (equivalent) example Display Names for the New Moniker are shown here:

```
new:pubsboserver.book
new:{CF166FD0-958C-11D3-A0F5-00A0CC330C70}
```

You could have created and bound a File Moniker using the following call to `MkParseDisplayName` instead of using the `CreateFileMoniker` function:

```
    HRESULT hRes;
    IBindCtx *pBindCtx;
    IMoniker *pMoniker;
    IDispatch *pDispatch;
    ULONG lEaten;

    hRes=CreateBindCtx(0,&pBindCtx);
    hRes=MkParseDisplayName(pBindCtx,L"c:\\alan\\filemonikertest.doc",
         &lEaten,&pMoniker);
    hRes=pMoniker->BindToObject(pBindCtx,NULL,IID_IDispatch,
         (void **)&pDispatch);
```

Microsoft has even provided an API function called `CoGetObject` that encapsulates the preceding logic. `CoGetObject` is defined as follows:

```
HRESULT CoGetObject(LPCWSTR strDisplayName,BIND_OPTS *pBindOpts,
                                    REFIID riid,void **ppv);
```

`CoGetObject` creates a `BindContext`, calls `MkParseDisplay`, and then calls `BindToObject` on the Moniker to create a COM object. Listing 12.7 shows how you would use `CoGetObject` to implement the same logic shown in Listing 12.6.

**LISTING 12.7** *USING COGETOBJECT WITH THE FILE MONIKER*

```
void CMonikertestclientDlg::OnCoGetObjectButton()
{
    HRESULT hRes;
    IDispatch *pDispatch;
    _Application WordApp;
    _Document WordDoc;

    hRes=CoGetObject(L"c:\\alan\\filemonikertest.doc",NULL,
                    IID_IDispatch,(void **)&pDispatch);
    if (FAILED(hRes))
    {
       _com_error err(hRes);
       AfxMessageBox(err.ErrorMessage());
    }
    WordDoc.AttachDispatch(pDispatch);
    WordApp.AttachDispatch(WordDoc.GetApplication());
    WordApp.SetVisible(VARIANT_TRUE);
}
```

Notice how much simpler this code is.

You can also use the `CoGetObject` together with the Class or New Monikers as an alternative to `CoCreateInstance(Ex)`. You can create a new instance of an object using the Class Moniker as shown in Listing 12.8.

**LISTING 12.8** *USING COGETOBJECT WITH THE CLASS MONIKER*

```
void CMonikertestclientDlg::OnClassMonikerButton()
{
   HRESULT hRes;
   IClassFactory *pFactory;
   IDispatch *pDispatch;

   hRes=CoGetObject(L"clsid:{CF166FD0-958C-11D3-A0F5-00A0CC330C70}",
                    NULL,IID_IClassFactory,(void **)&pFactory);
   if (SUCCEEDED(hRes))
   {
      hRes=pFactory->CreateInstance(NULL,IID_IDispatch,
           (void**)&pDispatch);
      pFactory->Release();
      // Use pDispatch here…

      pDispatch->Release();
   }
   else
   {
      _com_error err(hRes);
      AfxMessageBox(err.ErrorMessage());
   }
}
```

The Class Moniker only lets you bind to the Class Object for a COM class. So in Listing 12.8, you request the `IClassFactory` interface on the class object and then call `CreateInstance` on the `IClassFactory` interface pointer to create the desired COM object.

The New Moniker lets you bind directly to one of the interfaces that a COM class supports. The following code shows how to instantiate a COM object using `CoGetObject` and the New Moniker.

**LISTING 12.9** *USING COGETOBJECT WITH THE NEW MONIKER*

```
void CMonikertestclientDlg::OnNewMonikerButton()
{
   HRESULT hRes;
   IDispatch *pDispatch;
```

```
    hRes=CoGetObject(L"new:pubsboserver.book",
                     NULL,IID_IDispatch,(void **)&pDispatch);
    if (SUCCEEDED(hRes))
    {
       // Use pDispatch here...

       pDispatch->Release();
    }
    else
    {
       _com_error err(hRes);
       AfxMessageBox(err.ErrorMessage());
    }
}
```

The `GetObject` function in Visual Basic is the equivalent of `CoGetObject`. Listing 12.10 shows how to use the New Moniker via `GetObject`. The code shown in Listing 12.10 is the Visual Basic equivalent of Listing 12.9.

**LISTING 12.10** *USING VB, GETOBJECT WITH THE NEW MONIKER*

```
Private Sub Command1_Click()
    On Error GoTo errHandler
    Dim Book As Object
    Set object = GetObject("new:pubsboserver.book")
    ' Use the object here...

    Exit Sub
errHandler:
    Call MsgBox(Err.Description)
End Sub
```

You're probably wondering how `MkParseDisplayName` (and `CoGetObject`) works. It's actually quite simple. When `MkParseDisplayName` receives a Display Name, like `new:pubsboserver.book`, it looks beneath the `HKEY_CLASSES_ROOT\` key in the registry for a key with the same name as the string to the left of the colon in the Display Name (*new* in this case). The CLSID for the Moniker associated with the Display Name is stored beneath a registry key with this name. Figure 12.36 shows the Registry key and the CLSID for the New Moniker. `MkParseDisplayName` then instantiates the COM class identified by this CLSID and requests the `IMoniker` interface. Then it creates a bind context and uses the bind context to pass the information on the right side of the colon to the `BindToObject` function of the Moniker. This is the additional information the Moniker needs to do its

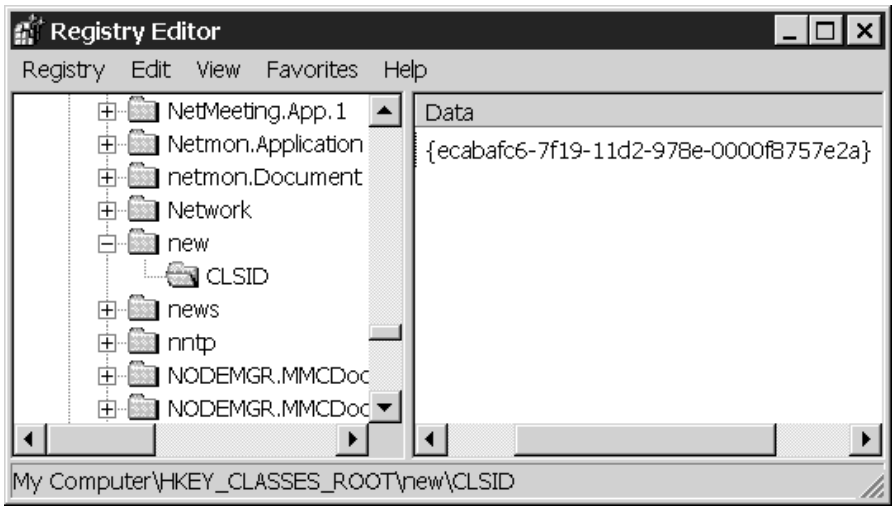



**FIGURE 12.36** The Registry Key and CLSID for the New Moniker.

work. You can create your own Monikers that use the registry and the `MkParseDisplayName` function.

### *The Queue Moniker*

Now that you understand a bit about Monikers in general, let's bring this discussion back to Queued Components and the Queue Moniker. To use the Queued Component Service, a client must instantiate the queued component using the Queue Moniker. If the client does not use the Queue Moniker (it uses `CoCreateInstance`, for example), the queued component is not queued (the method call is synchronous). This is actually good news; it means you can use queued components with or without the queuing service, depending on your needs at the time.

Although I have not seen the implementation of the Queue Moniker, it's not too difficult to imagine how it works. Instead of activating the queued object directly, the Queued Moniker must activate the QC Recorder and pass it the type library information for the requested object. The QC Recorder also has to know where to send its message when the QC Recorder is deactivated. The message the QC Recorder creates must also contain enough information for the QC player to instantiate the queued component on the server, so the Queue Moniker must also pass this information to the QC recorder. Figure 12.37 shows the Registry key and the CLSID for the Queue Moniker.

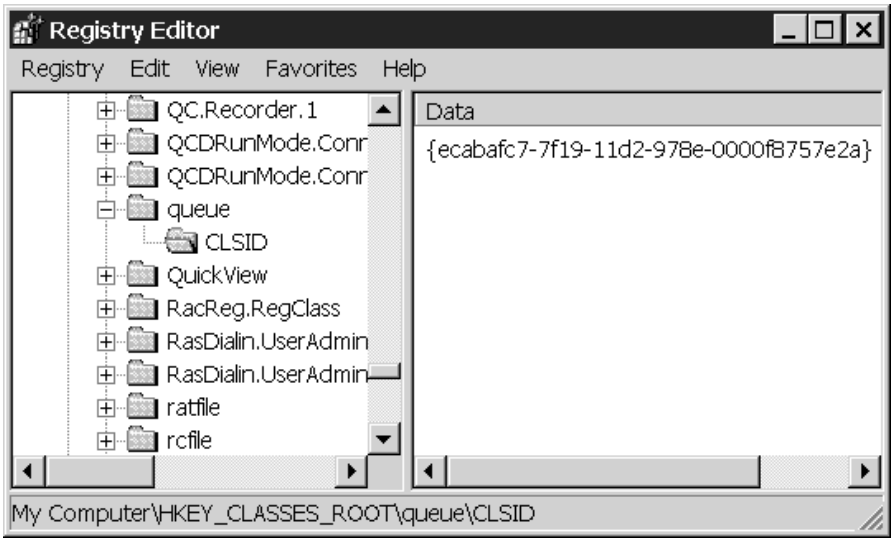



**FIGURE 12.37** The Registry Key and CLSID for the Queue Moniker.

The display name for the Queue Moniker has the following form:

```
Queue:[options]/New:CLSID
```

Notice there are actually two Monikers being used here: the Queue Moniker and the New Moniker. The Queue Moniker expects to receive the CLSID of the queued component from the new Moniker to its right. The following examples show how to use the Queue Moniker and the `CoGetObject` function to instantiate a queued component whose progID is `pubsboserver.bookprinter`:

```
CoGetObject(L"queue:/new:pubsboserver.bookprinter,
          NULL,IID_IDispatch,(void **)&pDispatch);
CoGetObject(L"queue:Priority=5,ComputerName=agordon1/" \
   Lnew:pubsboserver.bookprinter ",NULL,
   IID_IDispatch,(void **)&pDispatch);
```

From Visual Basic you can use the Queue Moniker as follows:

```
Set objPrinter = GetObject("queue:/new:pubsboserver.BookPrinter")
```

The Queue Moniker accepts a number of options that affect how it works; Table 12.8 gives the name and description of *some* of the parameters that can be passed to the Queue Moniker. See the MSDN library for the rest.

**TABLE 12.8** Parameters for the Queue Moniker

| Parameter | Description |
|---|---|
| ComputerName | The computer that the Queued Component service should send the message to. If this parameter is not specified, the ComputerName associated with the configured application is used. |
| QueueName | The name of the MSMQ queue that the Queued Component service should send the message to. If this parameter is not specified, the QueueName associated with the configured application is used. |
| PathName | The complete path of the MSMQ queue that the QC Recorder should use. This string is of the form ComputerName\QueueName. |
| AuthLevel | Should MSMQ authenticate messages using digital signatures?<br>MQMSG_AUTH_LEVEL_NONE<br>MQMSG_AUTH_LEVEL_ALWAYS |
| Delivery | Are MSMQ messages recoverable?<br>MQMSG_DELIVERY_EXPRESS (not recoverable)<br>MQMSG_DELIVERY_RECOVERABLE |
| Priority | A message priority level:<br>MQ_MIN_PRIORITY (0)<br>MQ_MAX_PRIORITY (7)<br>MQ_DEFAULT_PRIORITY (3)<br>Any number between 0 and 7 |
| PrivLevel | A privacy level, used to encrypt messages:<br>MQMSG_PRIV_LEVEL_NONE<br>MQMSG_PRIV_LEVEL_BODY<br>MQMSG_PRIV_LEVEL_BODY_BASE<br>MQMSG_PRIV_LEVEL_BODY_ENHANCED |

### *Poison Messages*

Once the QC Listener Helper receives a message on the server machine, it dequeues the message within the context of a transaction and then passes the message to the QC Player, which attempts to play the message back to the queued component. If one of the method calls fails or the transaction aborts for any other reason, the QC Listener Helper aborts its transaction, which returns the message to the queue. Since the message is now back on the queue, the QC Listener eventually tries to deliver the message again. If the queued component failed repeatedly, this could go on forever using up computing resources. This is called a *poison message*. MSMQ was designed to support high-availability and must strike a balance between handling poison messages and ensuring the system tries more than once to deliver a message to a queued component.

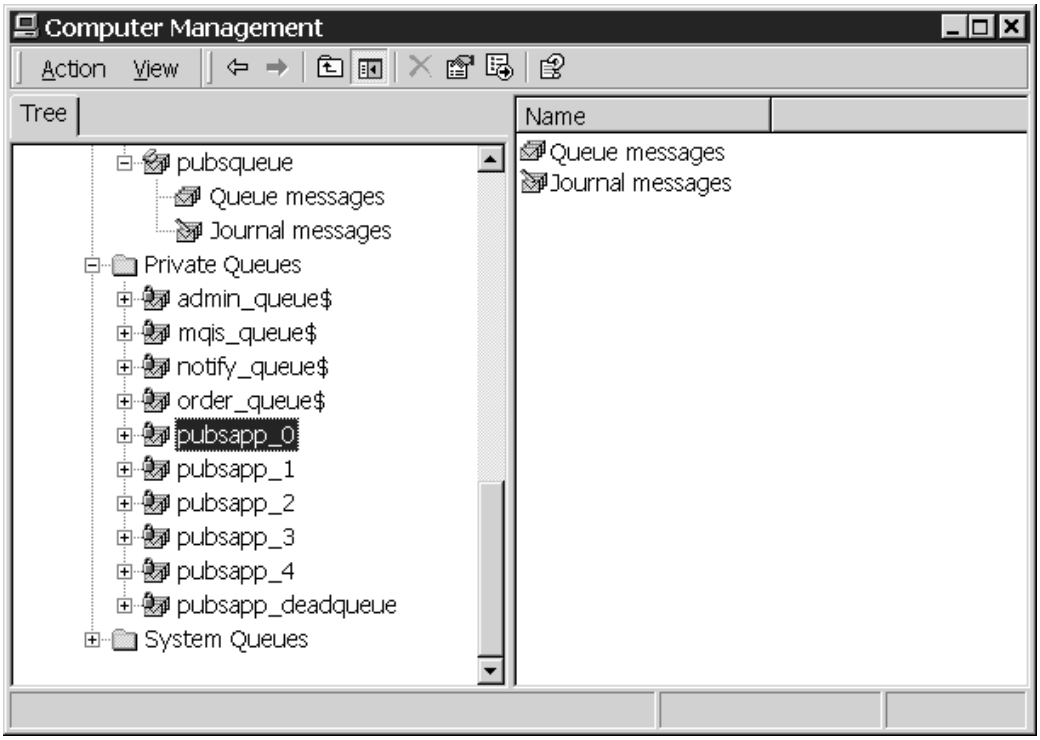



**FIGURE 12.38** The Retry Queues for a Queued COM+ Application.

The solution to this problem in the COM+ Queued Component Service is a series of six, private, retry queues. The COM+ runtime generates these queues automatically when you configure a COM+ application to support queuing. The retry queues for a COM+ Application are named as follows: there are five queues with the name `COM+ApplicationName_N` (where N = 0 to 4), and one queue with the name `COM+ApplicationName_deadqueue`. Figure 12.38 shows the private queues for a COM+ Application called pubsapp.

These retry queues are used as follows (this algorithm is subject to change by Microsoft at any time):

- **COM+ApplicationName_0**: If the transaction fails repeatedly on the main, public queue, the message is moved here. Messages in this queue are retried once each minute. After three unsuccessful attempts, the message is moved to the queue called `COM+ApplicationName_1`.
- **COM+ApplicationName_1**: Messages on this queue are processed every two minutes. After three unsuccessful attempts, the message is moved to the queue called `COM+ApplicationName_2`.

- **COM+ApplicationName_2**: Messages on this queue are processed every four minutes. After three unsuccessful attempts, the message is moved to the queue called `COM+ApplicationName_3`.
- **COM+ApplicationName_3**: Messages on this queue are processed every eight minutes. After three unsuccessful attempts, the message is moved to the queue called `COM+ApplicationName_4`.
- **COM+ApplicationName_4**: Messages on this queue are processed every 16 minutes. After three unsuccessful attempts, the message is moved to the queue called `COM+ApplicationName_deadqueue`.
- **COM+ApplicationName_deadqueue**: The final resting place. Messages remain on this queue until they are manually moved or purged using the Computer Management Explorer.

If you want to be notified when a message is being sent to the dead queue, you can register an exception handling class for your queued component. The exception class is just a configured COM+ class that implements the `IPlaybackControl` interface. You associate an exception class with a queued component by performing these steps:

1. Right-click on a component in a COM+ application.
2. Select `Properties...` from the `Context` menu (the component properties dialog appears as shown in Figure 12.39).
3. Click the `Advanced` tab (shown in Figure 12.39).
4. Enter the ProgID for the exception class in the Queuing exception class field.
5. Click `OK`.

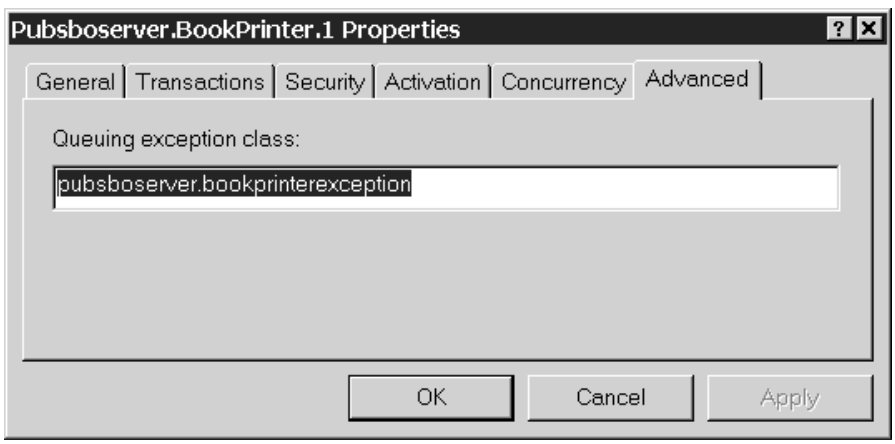



**FIGURE 12.39** Configuring a Queuing Exception Class.

The QC Service calls this exception class right before it sends the message to the deadqueue. At a minimum, this exception class should implement the `IPlaybackControl` interface. `IPlaybackControl` has two methods: `FinalClientRetry` and `FinalServerRetry`. The QC Service calls `FinalClientRetry` if (after executing the algorithm described above) it is unable to send a Message to the message queues on the server. The method is called right before the message is placed on the client-side deadqueue. In your implementation of `FinalClientRetry`, you can send an e-mail message to an administrator or write a notification to a log file or database. The QC Service calls `FinalServerRetry` on the server if (after executing the algorithm described above) it is unable to playback the message to the queued component. Once again, in your implementation of this method you can send an e-mail message to an administrator or write a notification to a log file or database. You can also attempt to complete the transaction.

Notice that `FinalClientRetry` and `FinalServerRetry` do not have any arguments. Your Exception class may want to know the value of the arguments that were passed to the queued component. To receive the arguments, your Exception must implement the queued interface that is being used. For instance, in Chapter 14 you will create a queued interface called `IBookPrinter`. If the Exception class wishes to see the values of the arguments that were passed to the `PrintCopies` method of `IBookPrinter`, then it must implement the `IBookPrinter` interface. The QC service replays the `PrintCopies` method to the exception class after it calls `FinalServerRetry`. The Queued Component example that you build in Chapter 14 won't use an exception class, but the Web site contains an enhanced version of this example that does use an exception class.

### The Role of MSMQ in COM+

Microsoft Message Queue (MSMQ), which was a part of the NT 4 Option Pack, lives on almost unchanged in Windows 2000. With Queued Components, I suspect that fewer people will use MSMQ in its raw form. Queued Components are the COM+ way to use MSMQ. As if to reinforce this, MSMQ is not administered in the Component Services Explorer. You administer MSMQ in the Computer Management Explorer as shown in Figure 12.33. Think of MSMQ as just a communication mechanism for COM+ Queued components. But you can use the Windows 2000 version of MSMQ in the same manner you used MSMQ on Windows NT 4. I explore MSMQ further when you build an example using queued components in Chapter 14.

## Load Balancing

COM+ was designed to facilitate the development of enterprise-class, distributed applications. In many cases, these applications need to support a large enough number of users or a processing load large enough that you cannot deploy your business objects on a single-server machine. You could solve this problem by adding multiple business object servers, but how do you divide the processing load between these servers? You could use a static load-balancing scheme. Perhaps all the people in one department could use one business object server and all the people in a different department could use another one. Aside from being more difficult to administer, this approach has a tendency to make inefficient use of your available computing resources. It is quite possible that one department taxes its server to the max, while another department's server is sitting idle most of the time. You would make far better use of your available resources if you used a dynamic load-balancing scheme. To make the most efficient use of multiple servers, you would like an activation request from *any* client to be automatically serviced by the server that currently has the lightest load.

You could implement this logic if you had the time and expertise. First you'd have to create a broker object that all clients must go to first when they want to instantiate a COM object. All the server machines that share the processing load for your application have to register themselves with the broker. The broker is then responsible for finding the server machine with the lightest load and instantiating the requested COM object there. Thanks to COM+, you do not have to implement this logic yourself. It is provided by the COM+ runtime and it is called the Component Load Balancing Service (CLBS). Although Microsoft's plans could change at any time, it appears you can only use CLBS if you have a server running the AppCenter Server version of Windows 2000 which is *not* scheduled to ship when the other versions of Windows 2000 ship in the first quarter of 2000.


**AppCenter Server is a version of Windows 2000 that will include deployment, management, and monitoring tools for Web applications running in server farms and will provide increased scalability and reliability for people who are deploying and managing high-volume, high-availability Web applications. The release version of AppCenter Server is not available as this book is going to press. However, CLBS was available in beta versions of Windows 2000 Advanced Server. The information and screen snapshots you see in this chapter are from Release Candidate 2 of Windows 2000 Advanced Server. Keep in mind that some of the information in this chapter may change by the time AppCenter Server is released, but the essence of the information should remain the same.**

The architecture of CLBS is shown in Figure 12.40.

To use CLBS, you have to designate one machine on your network as the load-balancing server as shown in Figure 12.40. This machine is the bro-

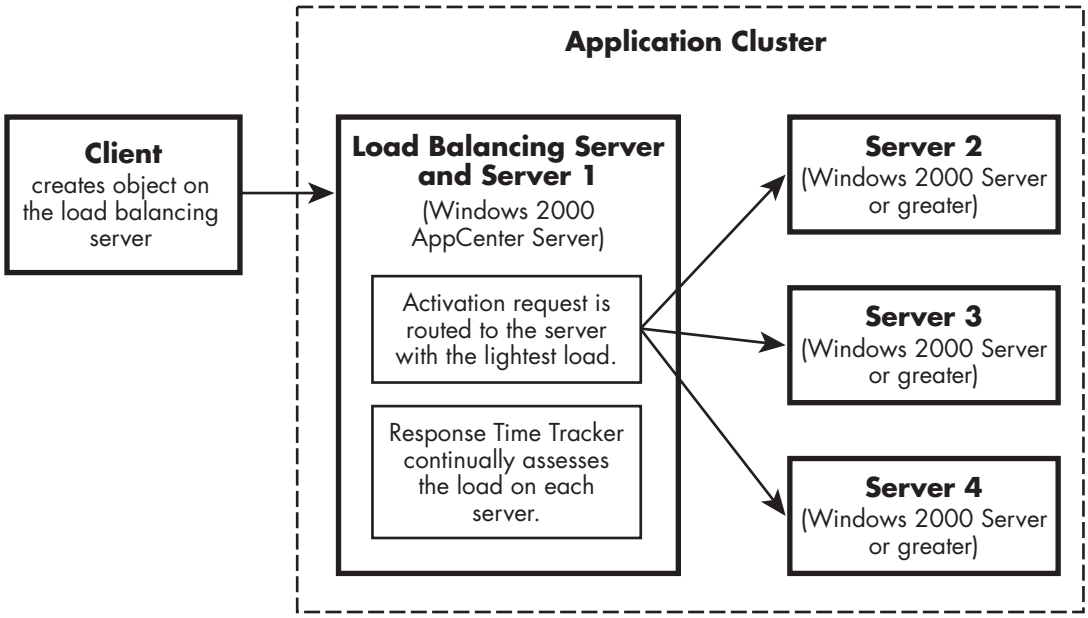



**FIGURE 12.40** The Architecture of the Component Load-Balancing Service.

ker that routes an activation request to the server that has the lightest load. This is also the machine that must be running AppCenter Server. Next, you must create an application cluster. This is just the group of machines that together dynamically share the load for your application. The load-balancing server itself is automatically made a part of this cluster and you can add other machines to the cluster manually. To configure a machine to be a load-balancing server and define an application cluster, perform the following steps in the Component Services Explorer:

1. Right-click on a machine in the Component Services Explorer as shown in Figure 12.41.
2. Select `Properties` from the `Context` menu (the computer properties dialog appears as shown in Figure 12.42).

3. Click the CLBS tab (Figure 12.43).
4. Set the Use this computer as the load balancing server checkbox.
5. Click the Add… button and add other machines on your network to the Application Cluster.

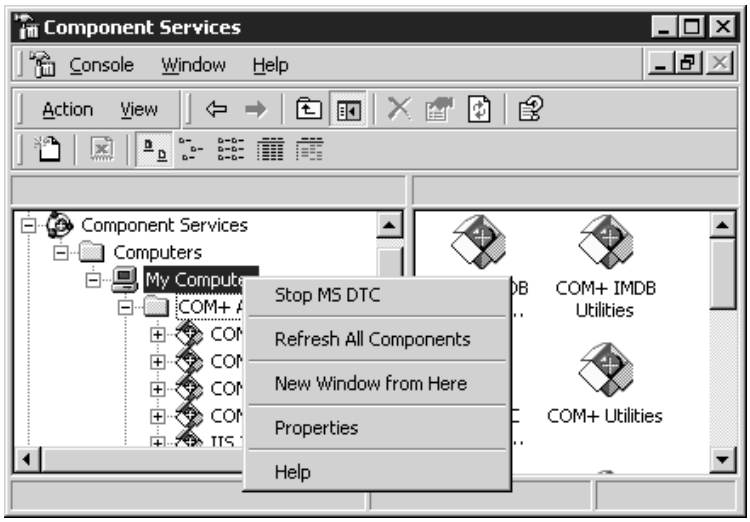



**FIGURE 12.41** Viewing the Computer Properties.

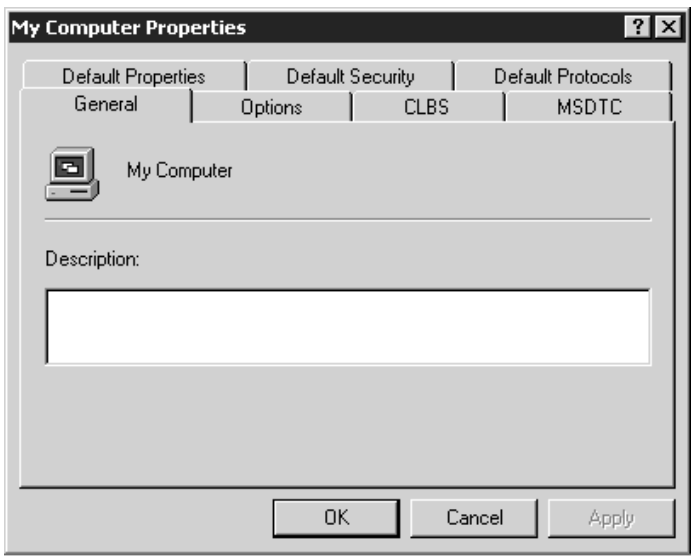



**FIGURE 12.42** The Computer Properties Dialog.

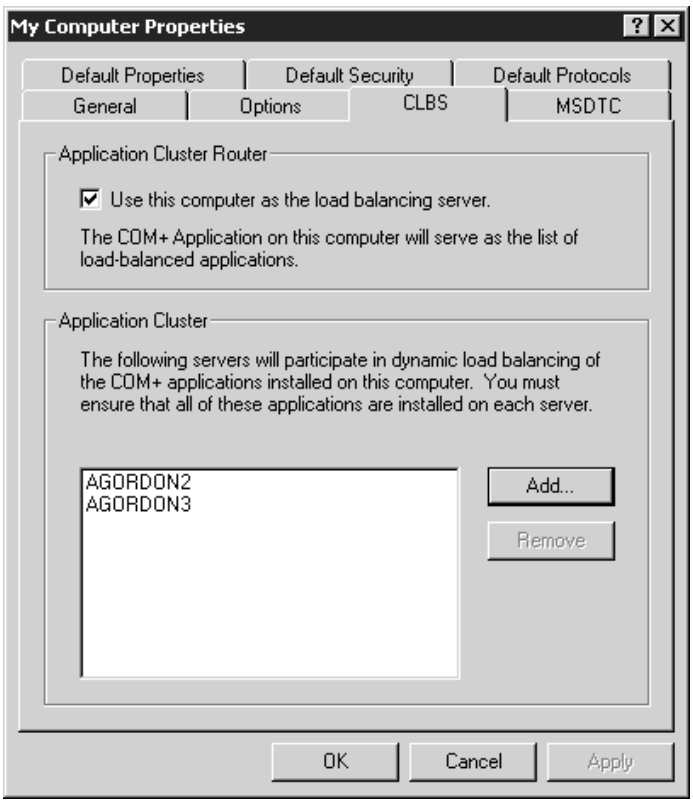



**FIGURE 12.43** The CLBS Tab of the Computer Properties Dialog.


**If you do not see the CLBS tab on the computer properties dialog, that means the machine you are using does not support CLBs.**

Once you have configured a machine to be your load-balancing server, you must also specify which components (classes) are allowed to work with the CLBS. To do this, perform the following steps in the Component Services Explorer:

1. Right-click on a component in a COM+ application.
2. Select `Properties...` from the `Context` menu (the component properties dialog appears as shown in Figure 12.44).
3. Click the `Activation` tab (shown in Figure 12.44).

4. Set the `Component Supports Load Balancing` checkbox.
5. Click `OK`.

Any classes you configure to work with CLBS must be location independent. By that I mean they should not have any dependencies that require them to run on a particular machine. Beware of hard-code directory paths (which are always a bad idea) and registry settings. Also, all software that the load-balanced components are dependent on (third-party software) must be installed on all machines in your cluster and (this should be obvious) the load-balanced component must itself be installed on all machines in the cluster. A good test you should always run prior to using the CLBS is to verify that the component you are using can be instantiated normally on each machine in your cluster.

All instantiation requests made on a load balanced component must be made on the machine that you have configured to be the load-balancing server. So if the load-balancing server is called agordon1, you should make all activation requests on that machine. The object may actually be instanti-

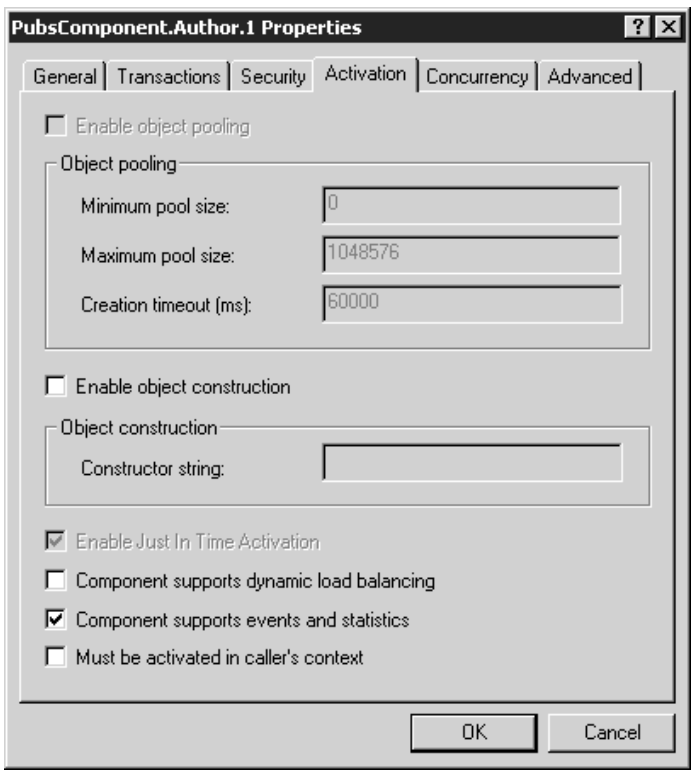



**FIGURE 12.44** Configuring a Component to Support Load Balancing.

ated on any machine in the application cluster, however. The load-balancing server runs a response-time tracker service. This service pings each machine in the application cluster and times how long it takes to receive a response. This is the load-balancing server's way of gauging the load on each machine in the cluster. The exact details of this algorithm are still up in the air and eventually you will be able to plug in your own algorithm. The load-balancing server maintains statistics that tell it which machine in the cluster (including itself) currently has the lightest load. It instantiates the object on the machine with the lightest load and returns a reference to the client. All communication after this occurs directly between the client and the machine on which the object was instantiated. The load-balancing server is not used as a go-between after the initial activation. The load-balancing mechanism is only used once: when the client first activates the object. If your component uses Just In Time (JIT) Activation, your object will *never* be instantiated on one machine in the cluster, deactivated because of JIT activation, and then re-activated on a different machine.

One of the best things about the CLBS is that using it is primarily an administrative task. An administrator has to setup the application cluster and then make sure your component is installed on each machine in the cluster. Aside from making sure that your components are location-independent you, the developer, really don't have to do anything special to allow your components to work with the CLBS.

## Compensating Resource Managers

A Compensating Resource Manager is any COM+ object that uses the Compensating Resource Manager (CRM) service provided by COM+. The COM+ CRM service is a set of tools provided by COM+ that make it simple to create your own resource managers. Unfortunately, the COM+ CRM service is another poorly named COM technology. Although you will often use *compensating transactions* to implement a CRM, this is not a requirement. A better name for this technology might have been the Resource Manager Framework, and a Compensating Resource Manager probably should just be called a Custom Resource Manager (you wouldn't even have to change the acronym).


See the sidebar "What is a Compensating Transaction"? on page 540 if you are not familiar with this term.

The COM+ CRM functionality lets you create your own resource managers with a minimum amount of effort. You're probably thinking why

would you want to create your own resource manager? Using custom resource managers, you can make any operation part of a transaction. For example, on the application I am working on we send billing information via XML to third-party accounting systems. When we send an XML file to the third-party accounting, we update tables in our database to indicate the bill has been exported.


This application uses MTS not COM+, but the problem is still the same.

In this use-case, two operations are performed: (1) you write an XML file to disk, and (2) you update the database to indicate that the XML file has been sent. Obviously, the database update is a transactional operation; it's rolled back if the transaction is aborted. But what about the writing of the XML file? This is not a transactional operation. The Windows 2000 file system is not a resource manager, although it eventually may be. This causes several problems for you. Imagine that you wrote the XML file first and then attempted to perform your database updates, but the updates failed. Your transaction rollbacks the database operations, but the XML file is still sitting on your file system. You could try reversing the order of the operations—performing the database updates first and then writing the file. This is better because if the database updates fail, you won't even attempt to write the XML file, and if the generation or writing of the XML file fails, you can just rollback the database transaction. There is still a potential problem with this approach, however. What happens if you encounter a system failure after you perform both operations? COM+ rollbacks the transaction (restoring the database to its state prior to the transaction). But once the file is written you can't take it back. You really need a COM+ aware resource manager to write the file to disk. It then becomes the resource manager's responsibility to ensure that the file is only written if the transaction commits. The COM+ CRM service provides all the tools you need to quickly and (relatively) easily build a resource manager like this. The architecture of a Compensating Resource Manager is shown in Figure 12.45.

It's easy to understand this architecture if you remember how the 2-phase commit protocol works. Remember that to work with the 2-phase commit protocol, a resource manager must perform its operations in two phases: (1) It must place the resources that it controls in such a state that it can guarantee it can make the changes permanent if the transaction commits, or undo the changes if the transaction is rolled back, and (2) it must wait for the transaction coordinator to inform it of the outcome of the transaction.

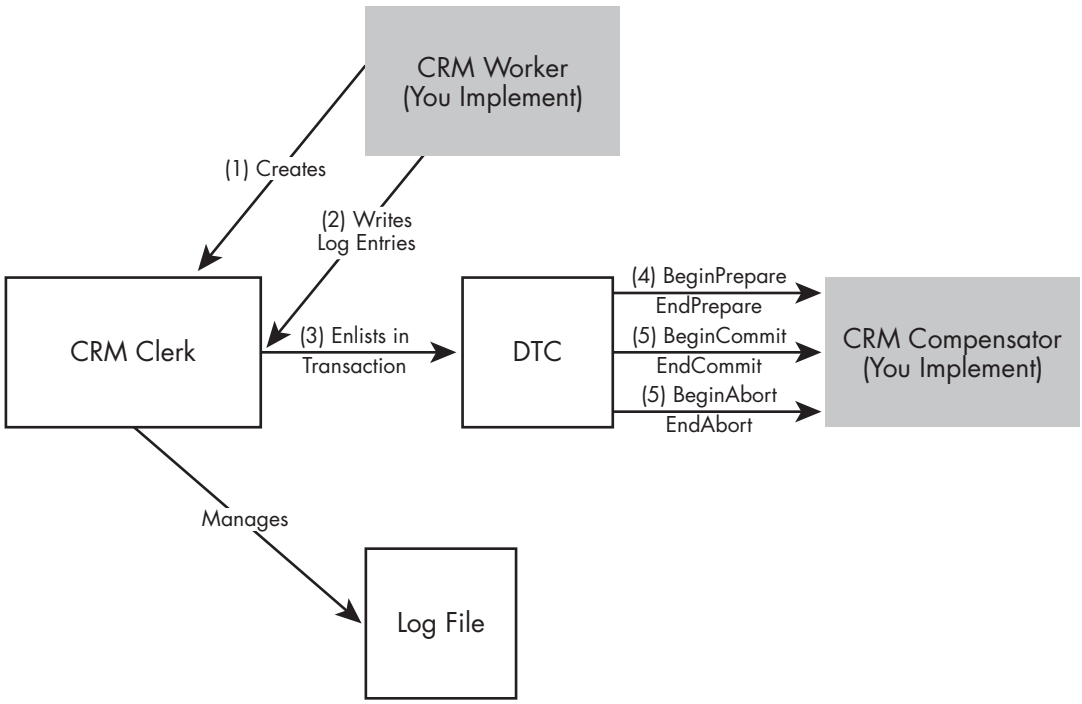



**FIGURE 12.45** The CRM Architecture.

Most resource managers implement this logic using a durable log file. In phase 1, the resource manager writes to the durable log file a record of all the work it intends to perform and then performs whatever work is necessary to place itself in a state that guarantees the outcome of the transaction. After these steps are performed, the resource manager can respond in the affirmative to phase 1 (see the sidebar entitled, "What's a Durable Log File" on page 539). What happens in the second phase depends on the outcome of the transaction. If the transaction commits, the resource manager must read the log file records and make the updates on its resources permanent. If the transaction aborts, the resource manager must read the log file records and undo whatever changes it made.

### What's a Durable Log File?

The log file used with a resource manager is called a *durable log file* because if the machines shut down abnormally (someone pulls the plug out of the wall, for example) on restart, the operating system (or transaction coordinator) reads the log file and checks to see if there are any transactions that did not complete in their entirety. If there are incomplete transactions, the operating system or transaction coordinator passes these log

records to the resource manager so it can try again to complete the operation. This is called *recovery*. The CRM support provided by COM+ implements a durable log file and recovery. Unfortunately, it does not automatically attempt recovery on system restart. The DTC does not attempt recovery until the COM+ application that contains the CRM is started. If recovery must occur on system restart, you should build your own mechanism to start the CRM's COM+ Application.

### What's a Compensating Transaction?

A Compensating Transaction is a transaction that reverses the affects of a previous transaction. For instance, if you buy an item using your credit card and then return the item, the merchant does not delete the original debit that was placed on your account. Instead the purchase price of the item is credited to your account, which cancels out the debit.

As Figure 12.45 shows, you must implement two COM components to create a CRM: a CRM Worker and a CRM Compensator. The CRM Worker contains the actual business logic that the CRM performs. So if your CRM writes XML files, your CRM Worker may likely contain a `WriteXML` method. The CRM Worker uses the services of the CRM Clerk component (provided by COM+) to register its CRM Compensator and to write records to a durable log file that the CRM Clerk maintains. When the DTC attempts to commit the transaction, it communicates with the CRM Compensator and call methods in the `ICRMCompensator` interface (that the CRM Compensator must implement) informing it of the outcome of the transaction. The DTC passes the log records written by the CRM Worker to the CRM Compensator so the CRM Compensator has all the information it needs to undo or make permanent the work depending on the outcome of the transaction.

When the CRM Worker is activated, it should create an instance of the CRM clerk. The CRM clerk implements the `ICrmLogControl` interface. The key methods in this interface are summarized in Table 12.9.

**TABLE 12.9** The Methods in the ICrmLogControl Interface

| Method Name | Description |
|---|---|
| RegisterCompensator | Registers the CRM compensator component that goes with this worker. |
| WriteLogRecord | Writes unstructured data to the log. You must call ForceLog before the records become durable. |
| WriteLogRecordVariants | Writes an array of variants to the log. You must call ForceLog before the records become durable. |
| ForceTransactionToAbort | Force the transaction to abort immediately. |
| ForceLog | Makes all log records that have been written durable so they can be retried if the machine shuts down abnormally. |

The CRM Worker should call the `RegisterCompensator` method on `ICRMLogControl` and pass in the ProgID of its CRM Compensator as shown in Listing 12.11.

**LISTING 12.11** *INITIALIZING A CRM WORKER COMPONENT*

```
HRESULT CCRMWorker::Activate()
{
   HRESULT hRes;
//  m_pCRMLogControl is a member variable declared as follows:
// ICrmLogControl *m_pCRMLogControl ;

   hRes = CoCreateInstance(CLSID_CRMClerk, NULL, CLSCTX_SERVER,
            IID_ICrmLogControl,(void **)&m_pCRMLogControl);

   hRes = m_pCRMLogControl-
>RegisterCompensator(L"Crmserver.MyCompensator",
            L"MyCompensator",CRMREGFLAG_ALLPHASES);

   return hRes;
}
```


**You won't build a CRM example in this book, but I have implemented a file writer CRM you can download from this book's Web site, and the code that you see in Listings 12.11 and 12.12 is from my file writer CRM.**

In its business logic, a CRM Worker should write its log file records and call `ForceLog` before it makes any system updates so that if the system crashed while it was making its updates, there would be enough information in the log to undo the changes. The CRM Worker should write enough information into the log so the CRM Compensator can undo or redo the operation. An example is shown in Listing 12.12. Some error checking is omitted from this code for clarity and brevity.

**LISTING 12.12** *PERFORMING WORK IN THE CRM WORKER COMPONENT*

```
1.  STDMETHODIMP CCRMWorker::WriteToFile(BSTR strText, BSTR strPath)
2.  {
3.     HRESULT hRetval;
4.     _bstr_t strDescription;
5.     IObjectContext *pObjectContext;
6.     _bstr_t path(strPath);
```

```
7.      _bstr_t text(strText);
8.      ofstream outputFile;
9.      BLOB blob;
10.     hRetval=CoGetObjectContext(IID_IObjectContext,
           (void **)&pObjectContext);
11.
12.     blob.pBlobData = (unsigned char *) ((char*) path);
13.     blob.cbSize = path.length( ) * 2 ;
14.     hRetval = m_pCRMLogControl->WriteLogRecord (&blob, 1);
15.
16.     blob.pBlobData = (unsigned char *) ((char*) text);
17.     blob.cbSize = text.length( ) * 2 ;
18.     hRetval = m_pCRMLogControl->WriteLogRecord (&blob, 1);
19.
20.     m_pCRMLogControl->ForceLog();
21.
22.     outputFile.open(path);
23.     if (outputFile.is_open())
24.     {
25.        outputFile << text << endl;
26.        outputFile.close();
27.        pObjectContext->SetComplete();
28.        hRetval=S_OK;
29.     }
30.     else
31.     {
32.        pObjectContext->SetAbort();
33.        strDescription=L"Could not open file";
34.        hRetval=E_COULD_NOT_OPEN_FILE;
35.     }
36.
37.     if (SUCCEEDED(hRetval))
38.        return hRetval;
39.     else
40.        return Error((LPOLESTR)strDescription,IID_ICRMWorker,hRetval);
41. }
```

On lines 12–18 in Listing 12.12, you write the path of the file that is to be written and the contents of the file to the log. This is enough information to undo the operation (you can just delete the file) or redo the operation (you have the file path and contents so you can recreate the file if you have to). On line 20, you call `ForceLog` to make the log records durable. Then, on lines 22–29, you can attempt to write the disk file. If you successfully write the file, you call `SetComplete` on `IObjectContext`.

Let's explore the implementation of a CRM Compensator. A CRM Compensator must implement the `ICrmCompensator` interface. The methods in this interface are shown in Table 12.10. The DTC calls the methods in this interface as it is attempting to commit the transaction.

**TABLE 12.10** The Methods in the ICrmCompensator Interface

| Method Name | Description |
|---|---|
| SetLogControl | Delivers an ICRMLogControl pointer to the compensator so that it can write additional log records. |
| BeginPrepare | Notifies the compensator of the beginning of phase 1 of the transaction. |
| PrepareRecord | Called once for each log record that has been written. |
| EndPrepare | Notifies the compensator of the end of phase 1 of the transaction. The (logical) return value allows the compensator to vote on the outcome of the transaction. |
| BeginCommit | Notifies the compensator that the commit phase (phase 2) of the transaction is about to begin. |
| CommitRecord | Called once for each log record. |
| EndCommit | Notifies the compensator that the commit phase of the transaction is about to end. |
| BeginAbort | Notifies the compensator that the abort phase (phase 2) of the transaction is about to begin. |
| AbortRecord | Called once for each log record. |
| EndAbort | Notifies the compensator that the abort phase of the transaction is about to end. |

The sequence of method calls depends on whether the transaction is being committed or aborted, or if recovery is in process. If the transaction is being committed, the sequence of method calls on the CRM Compensator is as follows:

(1) `SetLogControl`
(2) `BeginPrepare`
(3) `PrepareRecord` (called once for each log record written in the CRM Worker)
(4) `EndPrepare`
(5) `SetLogControl`
(6) `BeginCommit`
(7) `CommitRecord` (called once for each log record written in the CRM Worker)
(8) `EndCommit`.

If the client aborts the transaction, none of the prepare calls are made on the compensator. The sequence of method calls in this case is as follows:

(1) `SetLogControl`
(2) `BeginAbort`
(3) `AbortRecord` (called once for log record written in the CRM Worker)
(4) `EndAbort`

If the CRM Compensator is being called for recovery, you will see first a call to `SetLogControl` and then either a sequence of commit method calls (`BeginCommit`, `CommitRecord`, `EndCommit`) or abort method calls (`BeginAbort`, `AbortRecord`, `EndAbort`), depending on whether the recovered transaction was committed or aborted. You use these methods to implement the transactional logic of your CRM. In the `Prepare` method calls, you can verify that the system is in a state that allows the transaction to commit. In the `Commit` methods, you perform whatever processing is necessary to make your changes permanent. In the `Abort` methods, you should perform whatever processing is necessary to return the system to its state prior to the start of the transaction. For instance, my transactional file writer CRM deletes the file that was written by the CRM worker if the transaction aborts. If the transaction commits, it leaves the file alone.

You must keep some things in mind as you implement your CRM compensator. You cannot assume that the same instance of the CRM compensator that processes the set of method calls in the prepare phase will process the method calls in the commit phase. Each phase must be implemented so it is independent of the others. That's why `SetLogControl` is called at the beginning of each phase and the log records are passed to the CRM compensator in each phase. If a client attempts to commit a transaction and then someone pulls the power cord out of the wall or there is some other system failure in the commit phase, the `Prepare` method calls will not be repeated during recovery, the CRM compensator will only receive a set of `Abort` or `Commit` method calls.

Another point to keep in mind as you implement your CRM compensator is that the CRM architecture does not address *isolation*. Isolation (the "I" in ACID) means code that is running outside of the transaction should not see the results of the transaction until the transaction is complete. Usually isolation is implemented using locking. Your file writer CRM writes its file to disk in the CRM worker and then deletes it in the CRM Compensator, if the transaction aborts. But if some other piece of software tried to read the file after the CRM worker has performed its job but before the CRM compensator has had a chance to delete the file, the file could be read even when a transaction aborts. For the file writer, it is simple to fix this problem. Since you wrote the contents of the file to the durable log, you can delay writing the disk file until the transaction commits (the

`EndCommit` method). The CRM worker would just perform some validity checks (verify that the specified directory exists, for example) and then write the file path and contents to the durable log.

Another point to keep in mind is to make sure that the updates you make in the commit and abort phases are *idempotent*. An idempotent operation is one where the end result of the operation is the same even if the operation is performed several times. For instance, appending text to a file is *not* an idempotent operation. If you perform the operation several times, the text in the file is repeated several times. But opening and writing to a file in truncate mode so that its existing contents are discarded *is* an idempotent operation, because the file always contains the same contents regardless of how many times you perform the operation. The commit and abort phases in your CRM compensator must be idempotent because these operations may be performed many times.

After you have implemented your CRM Worker and Compensator, you must still perform some configuration to make them work properly. Your CRM worker and Compensator must be configured, so you must create a COM+ application and add both of these components to it. The COM+ documentation recommends that you configure your CRM Worker as shown in Table 12.11.

**TABLE 12.11** Configuration Settings for a CRM Worker

| Configuration Item | Setting |
|---|---|
| Transaction | Required |
| Synchronization | Yes |
| JIT | Yes |
| ThreadingModel | Apartment or Both |

Your CRM compensator should be configured as shown in Table 12.12.

**TABLE 12.12** Configuration Settings for a CRM Compensator

| Configuration Item | Setting |
|---|---|
| Transaction | Disabled |
| Synchronization | Disabled |
| JIT | No |
| ThreadingModel | Apartment or Both |

After you have configured both of the components, you must also enable CRM in the COM+ application. To do this, perform the following steps:

1. Right-click on the COM+ Application as shown in Figure 12.46.
2. Select `Properties...` from the `Context` menu (the Application properties dialog appears as shown in Figure 12.47).
3. Click the `Advanced` tab (shown in Figure 12.47).
4. Set the `Enable Compensating Resource Managers` checkbox.
5. Click `OK`.

If a COM+ server application has the Enable Compensating Resource Managers option selected, then the first time the COM+ application starts up it creates a CRM log file to be used by all CRMs in the server application process. The name of the log file is synthesized from the GUID that was assigned to the COM+ application when it was first created. You can find the log file in the `DtcLog` directory beneath your windows system directory. CRM log files have the extension crmlog.

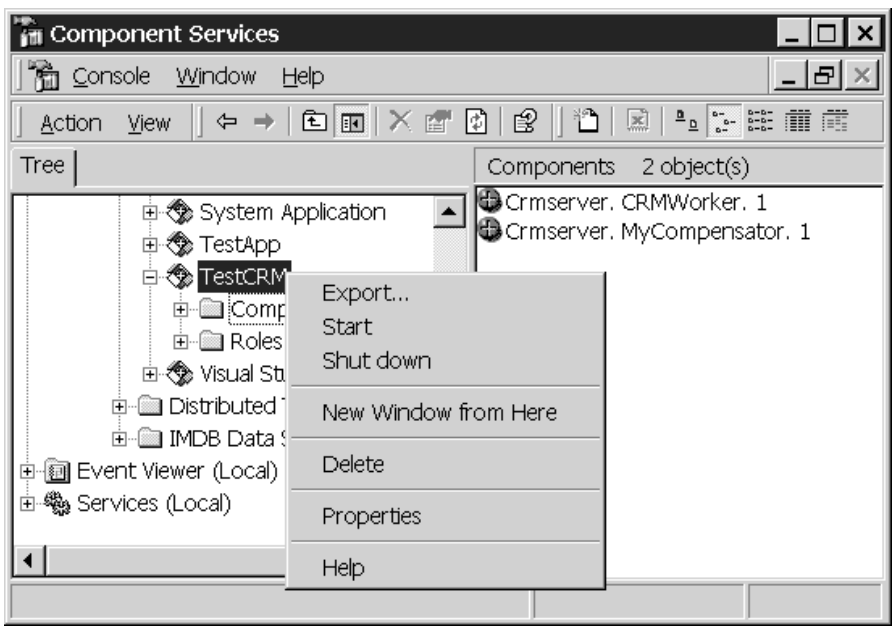



**FIGURE 12.46** The First Step to Enabling CRM.

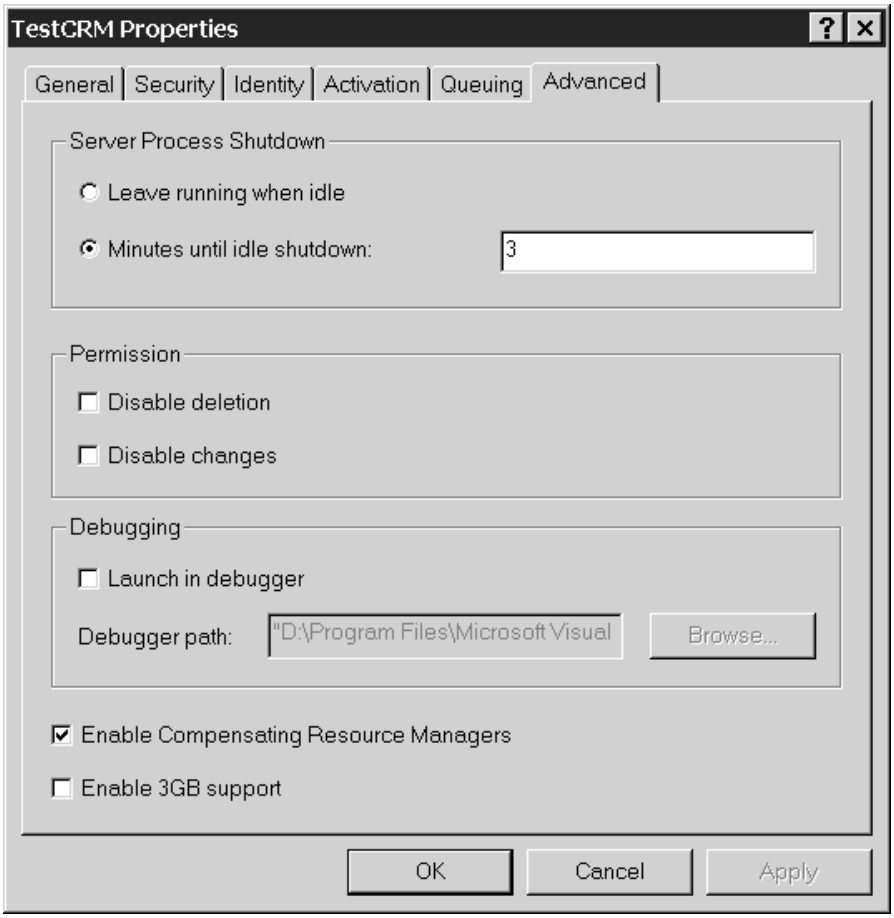



**FIGURE 12.47** Enabling Compensating Resource Managers.

## What Happened to the In-Memory Database?

In a nutshell, the In-Memory DataBase (IMDB) was removed from the release versions of COM+ and Windows 2000 because, as implemented, it would not meet the needs of most developers. It's likely that an improved version of this technology may find its way into later versions of COM+, but Microsoft has made no promises on this. Because it is not a part of the released version of COM+, I do not discuss it extensively, but since it will likely reappear sometime down the road, I thought it was at least worth a sidebar.

Anyone who has bought additional RAM for a PC knows about the amazing performance improvements you can gain from this upgrade. I recently upgraded my home PC from 128 to 256 MB of RAM and the improvement in performance was stunning. If you have a fixed

amount of money to spend to improve the performance of a computer, that money is usually best spent on RAM rather than getting a faster processor. Why? Because additional memory means the operating system can keep more applications and data in main memory instead of paging it out to disk. Main memory access is typically two or three orders of magnitude faster than disk access. This same rule applies to databases. Accessing database tables from a disk file is two or three orders of magnitude slower than accessing the database table once it is cached in memory. A significant performance improvement can be realized in any database application, regardless of how it is implemented, by caching frequently-used, read-only information into in-memory data structures. Of course you have to write code to query and cache the data, and you may also have to write some sort of indexing mechanism to make it easy and fast to extract information from the cache. The In-Memory DataBase (IMDB) was designed to do this work for you. Using IMDB, you could specify tables (you could not do sub-queries within tables and this was part of the reason for the cancellation) that IMBD should load into shared memory when its server process is started. This data is then made available to business objects or client code through an OLEDB provider (you learn what OLEDB and an OLEDB Provider is in Chapter 13). You could not perform SQL queries on the cached data through this OLEDB provider. You were only allowed to either browse the tables or use indexes to sort and filter. This was another major limitation that eventually caused IMDB to be canceled. A transactional version of the Stored Property Manager (the TSPAM) was also removed from COM+ at the same time that IMDB was.

## ■ Summary

In this chapter, you received an introduction to the COM+ services. I discussed each of the COM+ services in detail and you learned how to configure your COM+ component to support these services. In the next chapter, I put all this knowledge to use and build a complete, 3-tiered, thin-client application using COM+ and Windows DNA 2000.