Transactions

CHAPTER

6

IN THIS CHAPTER

Transactional design is crucial for a successful Online Transactions Processing (OLTP) application, and yet it is often totally "forgotten" in component-based applications. In this chapter, I discuss several different transaction techniques and recommend ways to choose between them. I then discuss transactions in the context of the proposed architecture of the previous chapter and show how you can design your application, making it easy to change transaction techniques if need be. Next, we look at the changes brought about by .NET concerning transactions and discuss tips on getting shorter transactions, lessening the risk of deadlocks, and avoiding traps with automatic transactions. Finally, I analyze my various transaction proposals based on the criteria established in Chapter 2, "Factors to Consider in Choosing a Solution to a Problem."

NOTE

This chapter discusses general solutions for transactions. In Chapter 8, "Data Access," I'll bind the solutions to my data access proposal and specifically to the architecture.

Locking and concurrency control are at the heart of transactions. We will touch on the subject in this chapter, but we'll also discuss locking and concurrency control in more detail in Chapter 9, "Error Handling and Concurrency Control."

Before we get started, it's important that you have a firm grasp of the following topics regarding general transaction theory and automatic transactions in .NET because I will not discuss them in detail in this chapter or anywhere in this book.

- Atomicity, Consistency, Isolation, and Durability (ACID)
- Shared and exclusive locks
- Transaction Isolation Levels (TIL)
- Two-Phase Commit (2PC)
- Doomed, done, and happy flags
- What the different transaction attribute values stand for

NOTE

If you feel you need to catch up on these topics, I recommend Tim Ewald's *Transactional COM+: Building Scalable Applications*[1] or Ted Pattison's *Programming Distributed Applications with COM+ and Visual Basic 6.0*.[2] To delve even deeper in general transaction theory, I recommend Jim Gray and Andreas Reuter's *Transaction Processing: Concepts and Techniques*[3] or Philip A. Bernstein and Eric Newcomer's *Principles of Transaction Processing*.[4]

Choosing a Transaction Technique

As you know, you can choose from several different techniques when dealing with transactions. In this section, I compare distributed transactions with local transactions and discuss the different approaches to working with those types of transactions, such as using transactional serviced components, ADO.NET, and stored procedures. First, let's start by looking at the main goal of any chosen transaction technique.

The Main Goal of Any Chosen Transaction Technique

As I mentioned in Chapter 2, the main goal of any chosen transaction technique is that it produces *correct* results when needed for certain scenarios. Keep this in mind when you read this chapter's discussions of performance, scalability, maintainability, and so on. Correctness is most important.

Although you may think that it is a given that correctness is extremely important, this notion goes one step further with transactions. Recall what I said in Chapter 1, "Introduction," about the new feature of COM+ 1.5 called process recycling. Although nobody likes a memory leak and we all try to avoid and/or try to find them, they're often not a large problem, even for critical Web sites—you just recycle the process once a day and nobody notices. However, if, at the very same Web sites, one transaction a day or a month produces an incorrect result leading to an inconsistent database, such leaks become disasters. The good news is that all the techniques I discuss in this chapter can be used to create correct transactions. Having said that, let's focus on issues of raw performance.

Description of the Selection of Transaction Techniques

It's possible to categorize transaction techniques in several ways. First, we can categorize them as being local (as are ordinary T-SQL transactions), being taken care of by one SQL Server instance, or as being distributed as 2PC transactions coordinated by Microsoft Distributed Transaction Coordinator (DTC). Transaction techniques can also be described as being automatic or manual. In automatic transactions, the desired transaction semantics are declared rather than programmed. In manual transactions, the transactions are controlled with explicit start and end statements.

NOTE

Don't confuse automatic and manual transaction techniques with the implicit and explicit transactions in, for example, T-SQL. As you probably know, when you do an `UPDATE` in SQL Server without first starting a transaction, the `UPDATE` will be wrapped inside an implicit transaction. If you begin and end your transaction on your own, you create an explicit transaction.

The final category is controller technology, such as ADO.NET (that wraps local T-SQL transactions) and the two wrappers for DTC transaction (namely, COM+ transactions and T-SQL distributed transactions). The last and very common transaction wrapper isn't really a wrapper. I'm referring to making pure T-SQL transactions. If we use the wrappers as categories, the situation shown in Table 6.1 occurs.

TABLE 6.1 Transaction Wrapper Techniques

Wrapper	*Manual/Automatic (Programmed/Declared)*	*Local/Distributed*
COM+ transactions	Automatic	Distributed
ADO.NET transactions	Manual	Local
Distributed T-SQL transactions	Manual	Distributed
Pure T-SQL transactions	Manual	Local

NOTE

In Table 6.1, you see that using COM+ transactions also means using distributed transactions. This is often overkill, especially if you have only one Resource Manager (RM) participating in the transaction. Even though a delegated commit will be used to optimize away some overhead from the 2PC protocol, this kind of transaction is expensive.

I haven't used distributed T-SQL transactions in any real-world applications, and I often find this to be a less commonly useful technique. Therefore, I will not discuss it in any detail now when describing wrappers. Instead, I'll briefly describe the different wrappers so that you understand what I mean when I use the different names. Note that I expect you to have a firm grasp of the techniques I present next so I will only describe them briefly.

COM+ Transactions

When COM+ controlled transactions are used, you get automatic and distributed transactions. COM+ asks Microsoft Distributed Transaction Coordinator (DTC) for help with the physical transaction, but COM+ will tell DTC when to start and when to end the transaction with the DTC-enabled RM(s). The transactional behavior is declared on the components with `transaction` attributes, and then COM+ uses interception to start and end transactions. If you use the `AutoComplete()` directive on the methods, you don't have to write any code to manage the transactions. You can see this in Listing 6.1, where a stored procedure is called and COM+ is starting and ending a transaction. Otherwise, you should use, for example, `ContextUtil.SetComplete()` and `ContextUtil.SetAbort()` to vote for the outcome.

NOTE

Although you obviously do not have to use stored procedures, I highly recommend it. In Chapter 5, "Architecture," I recommended that you always use stored procedures when components call the data tier. This presents a number of problems, but they are solvable. In Chapter 8, "Data Access," I present a few examples of such problems and offer suggestions for their solution.

LISTING 6.1 An Example of a COM+ Controlled Transaction

```
aCommand.ExecuteNonQuery()
```

NOTE

It's not only database engines that are RMs. Don't forget that Queued Components (QC) and MSMQ are other examples of RMs. In the future, I hope to see DTC-enabled RMs for Exchange's data storage, the Windows file system, and so on.

A major advantage with automatic transactions is that you can often reuse components without code changes, and they can directly participate in the transactions of the new consumers. One reason for this is that all the components participating in one transaction can open a connection of their own. The connections will auto-enlist in the transaction.

ADO.NET Transactions

ADO.NET transactions are conceptually really just a wrapper around ordinary T-SQL transactions. Listing 6.2 shows an example of how ADO.NET transactions can be used. In this case, there is a transaction around a call to a stored procedure. Of course, there is more to it than that—you have to make an `aConnection.RollbackTrans()` at the time of an exception.

LISTING 6.2 An Example of an ADO.NET Controlled Transaction

```
aTransaction = aConnection.BeginTransaction()
aCommand.Transaction = aTransaction
aCommand.ExecuteNonQuery()
aTransaction.Commit()
```

Pure T-SQL Transactions

Although T-SQL transactions are used by ADO.NET transactions, when I refer to "pure" T-SQL transactions, I'm referring to T-SQL transactions controlled by SQL scripts or by stored procedures. Listing 6.3 shows an example of how this might look. Once again, I have excluded the code for `ROLLBACK TRAN` and the complete code structure discussed in Chapter 5.

LISTING 6.3 Simplified Example of a Pure T-SQL Transaction

```
BEGIN TRANSACTION

UPDATE errand
SET closedby = @userId
, closeddatetime = GETDATE()
, solution = @solution
WHERE id = @id

INSERT INTO action
(id, errand_id
, description, createdby
, createddatetime, category)
VALUES
(@anActionId, @id
, @description, @userId
, GETDATE(), @aCategory)

COMMIT TRANSACTION
```

Why Care About Which Transaction Technique to Use?

There are huge differences between the different techniques when it comes to performance and scalability. The biggest difference is between local transactions and distributed transactions, because the 2PC protocol used by distributed transactions is pretty expensive as far as overhead is concerned. Table 6.2 presents a subjective overview of the advantages and disadvantages of each technique when you work with one RM. Note that the lower the value, the better.

TABLE 6.2 Comparison of the Transaction Techniques for One RM

Factor	*COM+ Transactions*	*ADO Transactions*	*Pure T-SQL Transactions*
Throughput	3	2	1
Participation in transactions together with other (unknown) components	1	3	3

TABLE 6.2 Continued

Factor	*COM+ Transactions*	*ADO Transactions*	*Pure T-SQL Transactions*
Getting portable code with regard to different database products	1	1	3
Fine-grained control of when to start and end transactions	3	2	1

ADO and Symmetric Multi-Processing Machines

A few years ago, I ran a test of COM+ transactions, ADO transactions (not ADO.NET), and pure T-SQL transactions with VB6 written components. I was puzzled when I saw that the COM+ controlled transactions had a higher throughput than the ADO controlled transactions. It took me a while to understand that it was only true on Symmetrical Multi-Processing (SMP) machines. When I turned off one of the two CPUs in my test application server, the throughput for the ADO controlled transactions actually increased and the resulting relation between ADO transactions and COM+ transactions turned out as expected. You will find the results from a test of the same techniques but in this new environment of .NET and ADO.NET on the book's Web site at `www.samspublishing.com`.

Reasons to Use Distributed Transactions

You may wonder why, given that they are so expensive, I am discussing distributed transactions at all. The reasons are simple:

- *There is more than one RM*—If you have more than one RM that must participate in the transactions, you should use distributed transactions.
- *Components that are "unaware" of each other coexist*—If you need to reuse a component that is out of your control or if you don't want to rewrite the component to fit into your architecture of local transactions, you can easily solve the problem with distributed transactions instead. This is also a useful technique for using legacy components in the .NET world.

Conclusion and Proposal: Choosing a Transaction Technique

My proposal for choosing a transaction technique is simple—use pure T-SQL transactions if you only have one RM; otherwise, use COM+ transactions. Also, use COM+ transactions if

you need transactions to span unknown components that are not all within your control. This is another situation where COM+ transactions shine.

Transactions in the Proposed Architecture

Transaction design is extremely important and was a key factor I evaluated when creating the architecture I described in the last chapter. In this section, I'll discuss how the transactions fit in the architecture. But before we do this, let's review the proposed architecture.

Review of the Proposed Architecture

Figure 6.1 presents the architecture for the sample application Acme HelpDesk, which you first saw in Chapter 5. Keep this figure in mind in the following discussion of what I consider to be important information about an earlier attempt of an architecture I made. The figure applies to both my old and new attempt.

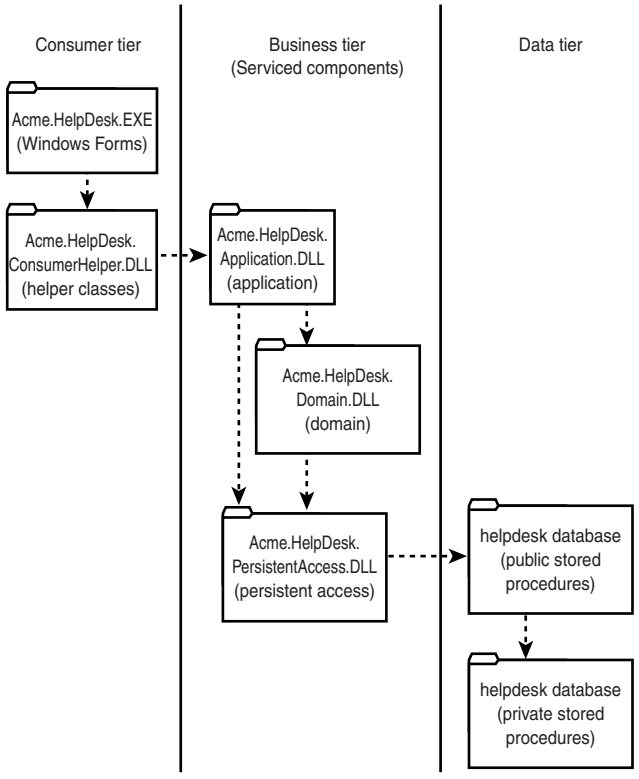

FIGURE 6.1

My architecture proposal applied to the sample application Acme HelpDesk.

Earlier Architecture Attempt

I've used several different architecture proposals over the years. One that I used a lot—let's call it the "old architecture"—looks exactly as shown in Figure 6.1, but there are a lot of differences to the current proposal. This old architecture isn't the last one I used before moving to .NET, but rather something that I used a few years ago. I'd like to discuss it here to show how my current architecture evolved.

When using automatic transactions in the old architecture, I only let the Persistent Access layer be transactional, so all the classes in the Application layer and the Domain layer had `NotSupported` for the `Transaction` attribute. (They could also have had `Supported` or `Disabled`, depending on what behavior was desired.) The idea was to get transactions as short as possible and not to let a transaction span more than one layer.

The main drawback was that using transactions only in the Persistent Access layer had a huge influence on design. All transactions had to reach one method in the Persistent Access layer in one call. Of course, this is possible, but it gave unintuitive code for the Application and Domain layers and it also made reusing the Domain and Persistent Access classes for different Application classes harder.

Another drawback when I used the architecture with manual transactions was that I relied on the stored procedures to control the transactions. The problem was when several stored procedures had to be called from a Persistent Access layer class to participate in one single transaction. In this situation, I let ADO control the transaction.

Yet another negative aspect about this proposal was that I split all the classes in the Persistent Access layer into two categories—one for fetching with the `Transaction` attribute set to `Supported` and one for writing with the Transaction attribute set to `Required`. Yet another drawback was that because I let the Persistent Access layer be transactional, I had interception and nondefault contexts for both the classes in the Application layer and the Persistent Access layer.

Transactions in the Current Proposal

When I use automatic transactions with the current proposal, I only declare transactional behavior on the classes in the Application layer by setting the `Transaction` attribute of the transactional classes to `Required`. (The classes in the other layers will have `Disabled` as the value of their Transaction attribute if they aren't `Shared`. If they are `Shared`, no Transaction attributes will be used, nor will inheritance from `ServicedComponent`.) Usually, the first contact with the database in a scenario will be deferred until the very end when the SQL script is to be executed. Consequently, the physical database transaction will not be longer by letting the Application layer be transactional.

Owing to this solution, the design will be less influenced by the transaction. The use case class in the Application layer can ask several Domain and Persistent Access classes for help by calling primitive methods, instead of having to move the control over to a Persistent Access class. The class residing in the Application layer will control the complete use case and, at the end, will send the SQL script to a helper class in the Persistent Access layer for execution.

Take a look at the interaction diagram in Figure 6.2 for an example of a call sequence. Here, you can see that the `ErrandSaving` class from the Application layer controls the scenario. First, it calls the `doErrand` class in the Domain layer to check whether the new errand is acceptable according to the defined rules. Then, it calls the `paErrand` class in the Persistent Access layer to receive the required rows to the SQL script. Finally, the SQL script is executed with the `paHelper` class and the public stored procedure `a_Errand_Insert()` is called. Finally, the private stored procedure `Errand_Insert()` will be called, and the `INSERT` statement will be executed there.

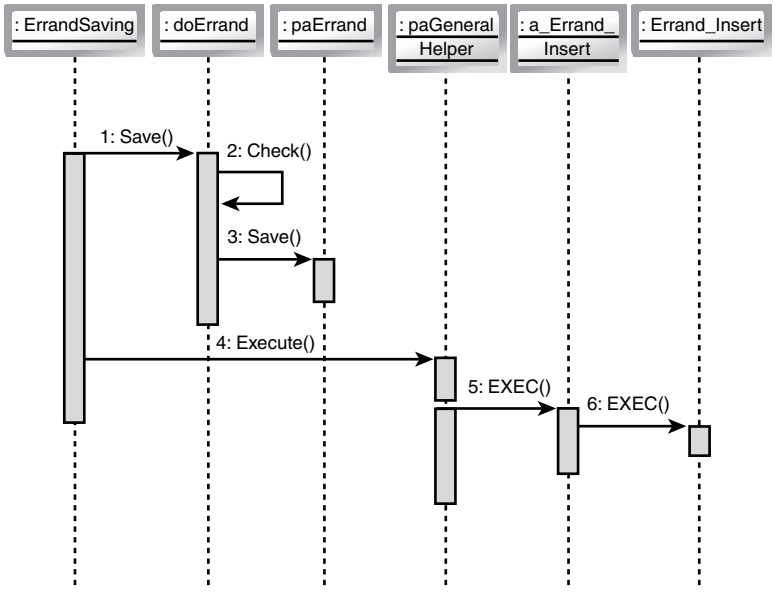

FIGURE 6.2
Interaction diagram, showing an example of a transaction.

Because an SQL Script is used, there is no need to use ADO.NET controlled transactions. Instead, the transaction can be started and ended in the SQL Script. There is no real reason for splitting the Persistent Access classes (`paErrand` in Figure 6.2) into two parts either.

The main drawback with the current proposal is that with automatic transactions, a method that does a fetch from the database and doesn't need a transaction will get a DTC transaction if it is located in a transactional Application layer class. When manual transactions are used, it isn't a problem to have a single class for both fetching and updating methods because I control exactly when to start and stop transactions. If it is a real problem with automatic transactions that fetching methods also starts DTC transactions, the classes in the Application layer have to be split. This is unfortunate because the use case then needs two different Application layer classes. Fortunately, the Consumer Helper layer can hide the fact that the class has been split from the Consumer layer.

A Flexible Transaction Design

If you follow the recommendation to use local transactions when you only have one RM, but want to prepare for a possible future change to distributed transactions and have as few programming changes as possible, there are a couple of things to think about.

Transaction Control

Assume you have followed the architecture that I have proposed. For simplicity's sake, we only have one root component in the Application layer that is called `ErrandSolving` in this scenario. It will use a component called `paHelper` in the Persistent Access layer. In turn, the `paHelper` will use a stored procedure called `a_Errand_Close()`. The stored procedure will `UPDATE` a row in the `errand` table and `INSERT` a row to the `action` table. Because there is only one RM involved in the transaction, I am using a local transaction in the stored procedure. Listing 6.4 shows how this may look, in a simplified version.

NOTE

I will discuss error trapping in greater depth in Chapter 9.

LISTING 6.4 Excerpt from Stored Procedure Showing a Simplified Pure T-SQL Transaction

```
BEGIN TRANSACTION

UPDATE errand
SET closedby = @userId
, closeddatetime = GETDATE()
, solution = @solution
WHERE id = @id
```

LISTING 6.4 Continued

```
INSERT INTO action
(id, errand_id, ...)
VALUES
(@actionId, @id, ...)

COMMIT TRANSACTION
```

Transaction Attribute Value

How should you declare the transaction-related attributes for the components in this situation when pure T-SQL transactions are used? There are several correct ways to do it. One possible proposal is to use the settings shown in Table 6.3.

NOTE

I assume here that `paHelper` isn't `Shared`. As you read in Chapter 5, it is preferable to use `Shared` whenever possible from a performance perspective.

TABLE 6.3 Transaction Settings: Proposal 1

Component	`Transaction(TransactionOption)`
`ErrandSolving`	`Supported`
`paHelper`	`Supported`

Because `ErrandSolving` is the root component for this scenario and uses `Supported` transactions, there will be no DTC transaction started. This is exactly the result I want. But did I achieve the result in the cheapest way? No, I can do better than this. A slight improvement would be to change `paHelper` to have `Disabled` instead of `Supported`. This makes it possible to save one context, but I'm still not satisfied. Let's investigate why not.

When `Supported` is used as the `Transaction(TransactionOption)` value, it means that you have to use just-in-time activation (JIT). As you recall from the discussion about JIT in Chapter 5, I prefer to use JIT only when I need COM+ transactions, so this isn't the perfect solution. Another way to see it is that because `Supported` requires interception, objects of this component can't go to the default context.

Before you say that you don't like this solution, remember that without any redeclarations, instances of both the components can participate in a COM+ transaction. If you really need the

components to be able to participate in a COM+ transaction, this solution isn't so bad after all. (It's not all that it takes, but it's the first step.)

A more efficient declaration would be as shown in Table 6.4, in which the components can still participate in an outer COM+ transaction.

TABLE 6.4 Transaction Settings: Proposal 2

Component	`Transaction(TransactionOption)`
`ErrandSolving`	`Disabled`
`paHelper`	`Disabled`

Now there will be no interception (if no instances of those components will co-locate in a context that uses interception). There is also the possibility of co-location that I discussed in Chapter 5. Less memory will be used and there will be less overhead for creation and calling methods.

The good thing is that if you add a new component that required a transaction, say `CustomizedErrandSolving`, a created instance of `ErrandSolving` can participate in the transaction. This assumes that instances of `ErrandSolving` must co-locate in the context of instances of `CustomizedErrandSolving`. We then have the best of both worlds.

The drawback is that co-location is often a problem for instances in the Application layer because that layer often requires component-level security, and then co-location is impossible. This is also the case for root components. If the consumer is Windows Forms on another machine, there is no context to co-locate in for the Application Layer class. We should still use the settings from Table 6.4, but instances of a new and, today, unknown component can't start a transaction in which instances of `ErrandSolving` can participate. In any case, we have declared transaction attributes as being as efficient as possible for the current situation. And we haven't created any obstacles so far for easily changing the transaction technique to COM+ transactions.

Changing the Values for Transaction Attributes

At first I thought I'd recommend the solution that I'm now in the middle of describing as a means for administrators to change transaction techniques when they needed to. When I mentioned this idea to Joe Long at Microsoft, he strongly disagreed, and we had a long discussion on the matter. His main concern was that administrators can't know about the inner workings of the components and the assumptions the programmers made when they wrote the code. Even for well and strictly architected components with full documentation of supported transaction settings, there is a risk that the administrator may make a small mistake. And we all know what one mistake can do to the transactions. The reason for using declarative

transactions isn't to make transaction semantics easier for administrators to maintain, easier for developers to program, or more efficient. The reason is correctness!

If we compare some of the `EnterpriseServices` attributes with other attributes in .NET, there is a fundamental difference. Some of the `EnterpriseServices` attributes can be changed from outside of the code. In the future, it will probably change, so we can lock the settings, but for the time being, it's very easy for someone to change the settings in the Component Services Explorer.

Another flaw with my original reason for my idea was that the administrator has little opportunity to make use of the flexibility I was about to propose. There are two basic situations when the transaction technique needs to be changed:

- The current components also need to hit another RM.
- The current components need to be used by other components, which in turn hit another RM, or they use another architecture for controlling transactions.

In the first case, it will usually be the developers who make that change anyway. In the second case, I think it's reasonable to contact the developers too.

Another reason—and certainly a big one—for not letting administrators change the settings in the Component Services Explorer for certain attributes is that the common language runtime won't look in the COM+ catalog, but rather in the meta data for the assembly to know how to deal with components regarding Object Pooling, for example. The only values that should be changed in the Component Services Explorer are deployment-related values, such as object construction strings, number of objects in the object pools, and similar deployment-related values—not the settings, for example, that enable object construction strings or object pooling.

So, Joe won the battle. I agree with him that setting the transaction attributes is a matter for the developers. Because of this, I have slightly changed the basic purpose of the proposal, and I'm now discussing it as a way for developers to prepare their design and code for a future change of transaction technique. They will carry out the change, but it will be easily done.

NOTE

It might sound strange to talk about the administrator as a person who would be thinking about using one of your components from another transactional component. However, it's very easy to publish XML Web services whose methods are transactional and that call your components. Meanwhile, BizTalk has been used to orchestrate a solution in which some of your components will participate. Think about this before you follow my recommendation of using `Disabled` for the transaction attribute. Are the implications reasonable in your situation?

Starting with Manual Transactions

Programming for a future change of transaction technique must start with coding for manual transactions and then move to automatic transactions, and not the other way around. Why? If everything works well for manual transactions, there is a good chance that it will work well for automatic transactions too.

On the other hand, if you start coding for automatic transactions, it's very common to use several connections in the same transaction so that participating components open their own connections. This works thanks to auto-enlist, so that all the connections will be enlisted in the automatic transaction. In a way, this is information hiding (which I normally appreciate) because the different components are more shielded from each other so they won't send around a connection object. On the other hand, this is less efficient than letting all instances that participate in the transaction share one connection. As you see, I prefer to let the instances share the connection because it's efficient and works with both automatic and manual transactions.

It's important to note that you shouldn't count on all RMs being able to successfully commit a transaction where several connections have been used in the same DTC transaction. This works well with SQL Server; however, at the time of writing it does not work so well with, for example, Ingres. (I've heard that a version of Ingres that is to be released soon—or has been released when you read this—will support transactions that span several connections, but it isn't documented as a requirement by Microsoft that "Tightly Coupled XA Threads" are a must for an RM to support DTC transactions.) A safe recommendation is to use only one connection in your DTC transactions too. On the other hand, be careful that you don't send a connection between processes or machines.

NOTE

You will see in Chapter 8 where I propose how to access the database, that only one method will hit the database for each scenario. The participating instances will just add logic to an SQL script that is executed against the database from a single method afterward.

There are two problems in starting with manual transactions and then expecting your components to work with automatic transactions without code changes. The first is that you have to remember that automatic transactions require JIT, so that the member state is lost when the transaction ends. What I mean is that you have to code your local transactions as if JIT were being used. That is definitely my intention with the architecture proposal.

The second problem is that you get slightly different transaction semantics in the two cases. You can delay transaction start with local transactions, and you will also, by default, work with another transaction isolation level (TIL). So watch out!

One Controlling Part

It's possible to use, say, `BEGIN TRANSACTION` and `COMMIT TRANSACTION` in your stored procedures, even if they are to be used from COM+ components. SQL Server will keep track of the current `@@TRANCOUNT` to determine whether `COMMIT TRANSACTION` really means a `COMMIT` or whether it is only subtracting one from `@@TRANCOUNT`. That's perfectly acceptable.

Nevertheless, there is at least one problem with this. `ROLLBACK TRANSACTION` won't just subtract 1 from `@@TRANCOUNT`; it will do a `ROLLBACK` of the complete transaction, which can create a problem for the components. In my opinion, it's much better to decide on just who is responsible for doing something, and then nobody else will interfere. (At least not as long as the first party does the job.) Therefore, I won't do a `BEGIN TRANSACTION` and `COMMIT TRANSACTION/ROLLBACK TRANSACTION` in my stored procedures if COM+ transactions are responsible for taking care of the transactions. However, this makes a real mess if the stored procedures are also to be used from other consumers that don't handle transactions on their own.

Moving from automatic to manual transactions for some scenarios will also take a lot of work. However, I use the solution to the problem shown in Listing 6.5. `@@TRANCOUNT` is stored in a local variable when the stored procedure is entered. When the stored procedure is about to start a transaction, it investigates whether there is already an active transaction. If there is, there won't be another `BEGIN TRANSACTION`. At `COMMIT/ROLLBACK` time, a similar technique is used. If there wasn't an active transaction when the stored procedure was entered, it should be `COMMIT`ted/`ROLL`ed`BACK` now. You should also add to this the criteria that there must be an active transaction. (There can now be several `COMMIT` sections in the stored procedure without creating any problems.) Clean and simple.

LISTING 6.5 Excerpt from a Stored Procedure Showing How to Write Flexible Transaction Code

```
SET @theTranCountAtEntry = @@TRANCOUNT
IF @theTranCountAtEntry = 0 BEGIN
  SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
  BEGIN TRAN
END

UPDATE...
```

LISTING 6.5 Continued

```
  --Another DML-statement...
  INSERT...

ExitHandler:
  IF @theTranCountAtEntry = 0 AND @@TRANCOUNT > 0 BEGIN
    IF @anError = 0 BEGIN
      COMMIT TRAN
    END
    ELSE BEGIN
      ROLLBACK TRAN
    END
  END
```

NOTE

Oracle, DB2, and even the SQL-99 standard do not support `BEGIN TRANSACTION`, but the first SQL command will start the transaction.

Something else you may need to add to this solution is handling the transactions that span over several stored procedures from your ADO.NET code. Then you can use `ContextUtil.IsInTransaction()` to determine whether you should start a new ADO.NET transaction. (You could also ask the database server for the `@@TRANCOUNT` value from your component, but that would lead to one more round trip, and you definitely don't want that.)

NOTE

Instead of starting transactions that must span several stored procedures from ADO.NET, I deal with this in the SQL script. This is at the heart of the data access pattern presented in Chapter 8.

Transaction Isolation Level (TIL)

If you use COM+ transactions, the Transaction Isolation Level (TIL) will be set for you. In the case of COM+ 1.5, you can configure the TIL you want to have. For COM+ 1.0, it will always be `SERIALIZABLE`. It's better to be safe than sorry.

NOTE

You can change the TIL within COM+ 1.0 transactions by using `SET TRANSACTION ISOLATION LEVEL` statements and optimizer hints. With SQL Server, this has a direct effect, but watch out because the behavior differs between different database products.

If you use manual transactions, you have to set the TIL on your own. (The default for MS SQL is `READ COMMITTED`.) I typically do this when I start the transaction seen in Listing 6.5. The problem is that, for instance, a public stored procedure doesn't know which TIL is needed by a used private stored procedure. It might then be the case that the public stored procedure `SET`s `REPEATABLE READ`, but the private stored procedure needs `SERIALIZABLE`. In this case, the private stored procedure must have its setting outside of the `IF` clause, so it executes even if the private stored procedure won't start the transaction. See Listing 6.6 for an example. When the code in Listing 6.6 executes, there is already an active transaction (`@theTranCountAtEntry` is not 0, so the `BEGIN TRANSACTION` won't execute), but the TIL will still be increased.

LISTING 6.6 Increasing the TIL Within a Transaction

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
IF @theTranCountAtEntry = 0 BEGIN
  BEGIN TRANSACTION
END
```

As you understand, it's very dangerous to change the TIL in code down the call stack, but as long as you only increase it, it's usually acceptable. How can you know that you have only increased the TIL? You can use `DBCC USEROPTIONS` and find an entry called `isolation level` that tells you the current TIL. If you don't find that entry, the TIL hasn't been changed and it has the default value. On the other hand, to avoid using this for my code, setting the TIL only to `SERIALIZABLE` can be done outside the `IF`-clause, as shown in Listing 6.6.

To summarize, I use the following rules for my stored procedures:

- If a transaction is needed, it will be started, but only if there isn't a current transaction.
- TIL is not `SET` if a transaction is already started, except if `SERIALIZABLE` is the needed level. That is, `SERIALIZABLE` will be `SET`, even if there is a current transaction.

I use the following rules when I generate the SQL script in the components:

- If the SQL script only calls one stored procedure, a transaction is not started from the SQL script. (The stored procedure starts a transaction if it is needed.)
- If the SQL script calls more than one stored procedure, the SQL script starts a transaction (if it is needed). Unfortunately, the only solution here is that the involved stored procedures must be investigated to decide whether a transaction is needed and should be started from the SQL script and what TIL is needed. This can be troublesome, but there is no shortcut.

As you know, I love to centralize code, so I tried to write helper stored procedures for my `BEGIN TRANSACTION` block and my block for ending transactions. This would have given me cleaner and smaller stored procedures, and I could also have hidden implementation details, such as that only `SERIALIZABLE` should be `SET` even if a transaction isn't to be started. Unfortunately, I failed because SQL Server monitors that the `@@TRANCOUNT` must have the same value when you exit a stored procedure as when you entered it. Otherwise, there will be an error.

NOTE

A colleague of mine once said that using a high (and correct) TIL is important if you work with bank applications, but you can cheat a bit and decrease it for simple administrative applications. This may sound like a dangerous viewpoint, but don't forget the context. In some applications, a high TIL may be very expensive and, at the same time, unnecessary. A typical example is applications that generate statistical information.

If you are unsure about which TIL to use for a certain scenario, go for a higher one, presumably `SERIALIZABLE`. Of course, you could standardize on `SERIALIZABLE` for all scenarios in the application, but I prefer to use the lowest correct TIL for each scenario.

`AutoComplete()` Versus `SetComplete()`/`SetAbort()`

As you know, with COM+ transactions, you must vote on the outcome. Should it be `COMMIT` or `ROLLBACK`? Even in this case, you have to pay extra attention to support the flexibility pattern that I discuss in this section. You can check `ContextUtil.IsInTransaction()` before you do a `SetComplete()` and `SetAbort()` to determine whether there is a transaction for which outcome you may choose. If not, there's no need to vote, and remember that `SetComplete()` and

`SetAbort()` don't only vote, they also set the done bit to `True`, which will lead to a deactivation of the object. Because I only use JIT for classes that use COM+ transactions, I don't want the deactivation to occur in other situations.

Another solution to the problem of voting for what transaction outcome you want to have is to use the `AutoComplete()` attribute on the methods for the transactional classes instead. With `AutoComplete()`, a raised exception is understood as a `SetAbort()`, but you don't have to write any code for it. At the same time, a method that ends without a raised exception is thought of as being `SetComplete()`.

A somewhat subtle positive effect of `AutoComplete()` is that you don't have to have a `Catch` block just to get a place for your `SetAbort()`, as shown in Listing 6.7. In this case, you can just skip the catch block because nothing is really happening except the `SetAbort()`.

LISTING 6.7 Catch Block Only Due to Calling `SetAbort()`

```
Catch e As Exception
    ContextUtil.SetAbort()
    Throw(e)
```

A problem with `AutoComplete()` is that if you get an exception but want to make a compensating action in a method higher up in the call stack and still `COMMIT` the transaction, it is impossible if `AutoComplete()` has already voted for the secondary object and deactivated it. In this case, the transaction is doomed. (Of course, in this situation, `SetAbort()` in the secondary method would give exactly the same result. `DisableCommit()` should be used instead.) Anyway, I usually prefer to take the easy way. If there is a problem, the transaction should be rolled back, and, because of this, the problem doesn't exist with `AutoComplete()`. Furthermore, if your secondary object co-locates in the context of the root object, there is no problem in the first place because the doomed flag isn't set until the context is left. As a matter of fact, it's not a good idea at all to let your secondary co-located instances vote. It's better to only let one instance in the context be responsible for the voting, preferably the root.

New Possibilities to Consider with .NET

.NET brings us a flood of new possibilities, although not all of them are even close to optimal. In this section, I point out a few weaknesses that the marketing department in Redmond doesn't talk much about.

Transactions and XML Web Services

A method on an XML Web service can use an automatic transaction (owing to a parameter of the `WebMethod` attribute). That transaction can span several .NET objects, but it cannot span

several XML Web services. The transaction won't flow. At first this may seem like a great limitation, but it makes sense because of the following reasons:

- Round trips between computers are always expensive. When XML Web services are used, it is not because they are the most performance efficient way of communicating, it is because of other reasons. What I mean is that transactions spanning several calls to XML Web services would be longer than with other communication mechanisms, and you don't want to have long transactions.
- Typical protocols for distributed transactions, such as OLE transactions (as are used by DTC), are inherently connection oriented. XML Web services are not.
- You don't know what technique is "hiding" behind that other XML Web service. Is it one that understands OLE transactions, for example? There is nothing in the XML Web services standard regarding transaction support.
- Several XML Web services publishers would certainly be very reluctant to let the consumers decide on the length of the transactions.

NOTE

Of course, you should pay a lot of attention to ensuring all the necessary information is given to the XML Web service in one method call so that it can take care of the complete transaction the normal way. This is a "must" for all root components in the Application layer, even if they aren't to be published as an XML Web service.

For the moment, the way to proceed to get transaction semantics over several XML Web services is to use a compensating mechanism instead. It won't be possible to fulfill the ACID properties, but this is the best you can do. Unfortunately, the Compensating Resource Manager (CRM) won't help you in this situation, even though its name suggests that it will. You have to roll your own solution instead.

The Compensating Resource Manager (CRM)

The Compensating Resource Manager (CRM) is relatively unknown, even though it has been around since COM+ 1.0 first saw the light of day. CRM helps to write a transactional resource dispenser of a resource that isn't DTC-transactional. CRM is to be considered as yet another RM. Typical examples of resources that the CRM are useful for dealing with are the file system and Microsoft Exchange. CRM won't perform magic and create a truly transactional resource dispenser for you, but with the help of compensating actions, you can go a long way.[5, 6]

NOTE

There is more and more interest in using sagas for transactions. A *saga* is a logical transaction aggregated from a sequence of physical transactions that must all succeed or be undone. One reason for the new interest in sagas is the interest in XML Web services. It is not within the scope of this book to discuss sagas further. For more information about sagas, see *Transaction Processing: Concepts and Techniques*[3] and *Principles of Transaction Processing*.[4]

Flow of Services Through Remoting

As you recall from Chapter 5, component services won't flow through XML Web services nor through Remoting. Actually, the lack of flow of component services through XML Web services is usually not as large a problem as it may seem at first. Most often, for reasons of efficiency, you have all the serviced components that are to talk to each other at the same application server (and in the same AppDomain). There is then no need for flow of component services over Remoting.

On the other hand, if you do need to let component services flow between machines, you can always rely on good old DCOM for that. Yes, DCOM is a nightmare through firewalls, but why would you have a firewall between your serviced components? If you do have a firewall between your serviced components, ask yourself again if this is the correct design to use.

Tips on Making Transactions as Short as Possible

The length of transactions affects scalability. Each transaction holds on to resources, such as locks in the database. If you can shorten your transactions, you can service more users (and transactions) with the same hardware. The following are tips I think are important in making transactions as short as possible.

Avoiding Large Updates

If you have to update several rows in a transaction, the transaction will take longer than if only a few rows are to be updated. No rocket science here. Even so, it is worth thinking about because, for example, this may affect your batch processes, which often update thousands of rows in each transaction. It is increasingly so that you don't have any downtime when you can run batch processes like this without interfering with other transactions. Therefore, it may be important to use a strategy other than using huge transactions. For example, you could use a compensating solution instead so that if you have to update 100,000 rows, you can update them in chunks of, say, 1,000 in each transaction. If one transaction fails, you will see that the

total operation isn't atomic. As a result, you will have to compensate for this. For example, try again with those transactions that weren't updated the last time, or undo the result of the transactions that were updated before. Watch out—these can be dangerous design changes to make, and you have to make careful evaluations before moving along.

NOTE

Note that batch processes don't go well with COM+ transactions. First we have the timeout, saying that the complete transaction may not take more than x seconds. (Normally in production, x is set to 5 or lower.) Then we have the JIT behavior that leads to longer execution time, without any real use in this situation. Distributed transactions are also a drawback because they will be more resource consuming, which is not wanted if you don't need distributed transactions.

Another typical situation arising from too large updates is when you haven't used a high enough normal form. It then might be the case that a certain value is located in thousands of rows and, when the value has to be updated, you have a very large transaction to deal with. The solution to this is simple—use a high enough normal form for your database design, usually the third normal form or higher.

Avoiding Slow Updates

For all tasks, there are a number of different approaches to use. For relational databases, the correct way to proceed is usually to use a set technique instead of a row technique. Let's take a simple example. If you are going to increase the price of all products for a certain category in a stored procedure, you can do this by opening a `CURSOR` with a `SELECT` that fetches all the rows to `UPDATE`. Then you iterate the `CURSOR` and `UPDATE` row by row. It works, but it will be much slower than a simple `UPDATE` that updates all the rows in one statement. This might be obvious, but you should think in this way more often and for less obvious scenarios too. Think twice if there is a loop in your T-SQL code—check that it isn't a design bug.

Loops

Although you want to avoid loops, you may sometimes need to use a loop after all. Typical reasons for this are

- You need to update a large number of rows and you don't want to fill the transaction log, or you don't want to lock out all other users from the table.
- You need to use a very tricky algorithm that can't be solved or that can't be solved efficiently with set techniques. (However, don't give up too fast.)
- You need to call a stored procedure for each row in a resultset.

Even though you need a loop, you don't have to use a CURSOR. As a matter of fact, I recommend that you use a WHILE loop instead. It's as fast as or faster than a CURSOR in almost all situations. (If the used key is a composite of more than two parts, a CURSOR is slightly faster.) The code is also simpler and cleaner with a WHILE loop, and there is less chance of making mistakes, but once again, if the key is a composite, the WHILE code is more complicated than the CURSOR code.

Let's look at an example of what a WHILE loop looks like in action. I will solve the same example that I just used (increasing the price of all products), but this time I will update all the products with a WHILE loop, and I will update them category by category. The products are split into ten different categories.

In Listing 6.8, you can see that I first SELECT what is the smallest category from the product table. If I found a category, the WHILE loop is started. I UPDATE all the products for the particular category, and I investigate that the UPDATE went all right as usual.

LISTING 6.8 WHILE Example: Part 1

```
SELECT @aCategory = MIN(category)
FROM product
WHILE @aCategory IS NOT NULL BEGIN
  UPDATE product
  SET price = price + @add
  WHERE category = @aCategory

  SELECT @anError = @@ERROR, @aRowcount = @@ROWCOUNT
  IF @anError <> 0 OR @aRowcount = 0 BEGIN
    SET @anErrorMessage = 'Problem with...'
    IF @anError = 0 BEGIN
      SET @anError = 81001  --An example...
    END
    GOTO ExitHandler
END
```

In Listing 6.9, you can see that I save the last processed category in an old variable, and then I search for the smallest category larger than the last processed.

LISTING 6.9 WHILE Example: Part 2

```
  SET @aCategoryOld = @aCategory

  SELECT @aCategory = MIN(category)
  FROM product
  WHERE category > @aCategoryOld
END
```

NOTE

Several years ago when I was porting an application that I built for SQL Server 4.21 to SQL Server 6, I decided to rewrite a `WHILE` loop to a `CURSOR` solution. I guess I was tricked by all the hype about server-side `CURSOR`s that was taking place at the time. It was probably not a good idea, especially because the `WHILE` loop worked just fine and there weren't any problems with it. (Well, there weren't any problems with the `CURSOR` either, but it was probably a worse solution.) Since then, I've learned not to make transitions like these without having a real purpose and without examining whether such transitions will have a positive effect.

Avoiding Pure Object-Oriented Design

As you recall from Chapter 5, I stress in this book that object orientation is great, but it has to be used wisely. I've seen pure object-oriented design used often for COM+ applications, and the result has typically been scalability that's too low. (I've not only seen it used, I've been contacted by e-mail and in person several times and asked what to do in these situations. My answer has always been "Redesign.")

NOTE

When I say pure object-oriented design, I mean classic object orientation, for example, with many properties. Each row and column from the database has lead to instantiated objects, and methods have been primitive so that to accomplish a task, several method calls must be made.

One of the reasons that pure object-oriented design doesn't scale is that the transactions will start too early, and because several objects will participate in the transaction and each one of them will talk to the database, there is also a lot of overhead for round trips. Usually, stored procedures are not used in those applications, and if they are used, it's only for primitive operations such as one `SELECT`, one `UPDATE`, one `INSERT`, or one `DELETE`.

It might seem compelling to have all the code in the components and not let the database and transactions affect the design at all. Unfortunately, it doesn't work well in large-scale situations. I've heard war stories about applications built this way not scaling beyond five to ten users.

Avoiding Pessimistic Concurrency Control Schemas

Another typical reason for having long transactions is the need to use pessimistic concurrency control schemas. Try to avoid pessimistic concurrency control schemas and you will get shorter transactions. Of course, the disadvantage is that you don't know whether your transaction will be able to run when you use an optimistic schema instead, but most often this is the way to go.

With Web-based applications, it's not usual to keep a connection for a user between page renderings. It's the same for all COM+ applications where the connection is closed and the result is disconnected from the database and sent back to the user. Because of this, a built-in pessimistic concurrency control schema can't be used. In any case, you never want a user to decide the length of a transaction by asking the user for an answer between start of the transaction and `COMMIT`.

NOTE

In Chapter 9, I will show you an alternative to the built-in pessimistic concurrency control schema that works in disconnected scenarios and doesn't create long transactions.

Using an Efficient Pattern for Data Access

I won't go into detail about my pattern for data access until Chapter 8, but it is crucial that you have an effective pattern for data access. As you can guess, I think my pattern is a very efficient one. I defer all access to the database until the end of a scenario. Until then, an SQL script is built with all the calls to different stored procedures. A transaction won't be started until it's needed in the SQL script. When the SQL script has started to run, there are no other round trips and no operations other than the calls to stored procedures until the transaction is to be ended.

Starting Transactions Late and Ending Them Early

Suppose that you have five tasks to accomplish. Quite often, only two of these need to be done inside the transaction, so you should program in this way, of course. Don't do anything in the transaction that you don't have to do. Prepare the transaction before it starts by getting `NEWID()`s, if you need to `INSERT` a row that has a `UNIQUE IDENTIFIER` as the key, for example. Listing 6.10 shows an example of a simplified stored procedure for reporting a new errand and, at the same time, writing an action because the reporter also made a first attempt to solve the problem that failed.

LISTING 6.10 A Simplified Version of a Stored Procedure for Inserting an Errand and a First Action

```
CREATE PROCEDURE a_Errand_Insert
(@userId uddtUserId, @errandDescription uddtDescription
, @actionDescription uddtDescription)
AS

  DECLARE @anErrandId UNIQUEIDENTIFIER
  , @anActionId UNIQUEIDENTIFIER
  , @aCategory UNIQUEIDENTIFIER

  SET @anErrandId = NEWID()
  SET @anActionId = NEWID()

  SELECT @aCategory
  FROM category
  WHERE description = 'report'

  BEGIN TRANSACTION

  INSERT INTO errand
  (id, description, createdby, ...)
  VALUES
  (@anErrandId, @errandDescription, @userId, ...)

  INSERT INTO action
  (id, errand_id
  , description, createdby
  , createddatetime, category)
  VALUES
  (@anActionId, @anErrandId
  , @actionDescription, @userId
  , GETDATE(), @aCategory)

  COMMIT TRANSACTION
```

NOTE

I have cut out the error trapping code from Listing 6.10. I will focus on error trapping in Chapter 9.

As you see in Listing 6.10, I waited until I had created the two GUIDs and I read the category GUID before I started the transaction. The difference isn't huge, but it's the principle I like to push. However, what happens if this stored procedure is called together with other stored procedures? There is a great chance that you will need to create an outer transaction, and then my recommendation here is of no use. Even if it won't matter in some cases, it will in others. This is also an indication that it might be better to let the components prepare as much as possible before starting the database transaction. In the example in Listing 6.10, this would mean that the GUIDs will be given as parameters instead. It might result in a little more network traffic because more data will be sent over the network, but it's usually worth it.

That was the usual recommendation. Just be careful that you don't overuse it so that you fetch rows from the database that will not keep their shared lock during the complete transaction because you had the fetch before the transaction started.

Using Correct TIL

Be careful that you use the correct TIL. Overusing `SERIALIZABLE` can definitely affect scalability. I can't tell you how many times I've heard developers at COM+ newsgroups ask why there is so much blocking in the database when they start using COM+. They aren't doing anything unusual. If you only execute a `SELECT COUNT(*) FROM mytable`, other transactions will not be allowed to `INSERT` rows `INTO mytable` until the first transaction is done. COM+ 1.0 always uses `SERIALIZABLE` for automatic transactions with SQL Server. You can configure the TIL for COM+ transactions in COM+ 1.5.

When you use pure T-SQL transactions, you should sometimes use `SERIALIZABLE`, sometimes not—you have to decide from case to case. You could go for `SERIALIZABLE` all the time, but then your transactions will be longer and you will also affect concurrency much more than if you use a lower TIL.

NOTE

For a good discussion on how to think about the TIL, see Tim Ewald's *Transactional COM+: Building Scalable Applications*.[1]

Avoiding 2PC with the Help of Replication

If you are going to use more than one RM in your transactions, you have to use distributed transactions if you like to have transaction semantics over the RMs. Distributed transactions will operate with the 2PC protocol, which is very expensive when it comes to performance

compared to local transactions. An alternate solution is to only `UPDATE` one of the RMs and then let the `UPDATE` affect the other RM with the help of replication. You won't get full transaction semantics, but often the consistency is good enough. In addition, the difference in throughput can be very big. This will also increase the reliability because if one of the participants in a 2PC transaction is down, the transaction can't be fulfilled. With replication, the transactions will still take place, even if one of the database servers is not operating at the time. The faulty server will get the `UPDATE`s when it comes back to life again.

Tips on Decreasing the Risk of Deadlocks

For local transactions, the RM will quickly find deadlocks itself. In this case, the RM will decide which transaction should lose, and it is interrupted so the other transaction can continue. Unfortunately, you can't catch a deadlock error in your stored procedures because they are interrupted, and even the batch that calls the stored procedure is interrupted. The error will always have to be caught in your components instead.

For distributed transaction, the Transaction Manager (TM) will, in most industrial implementations, not really try to detect a deadlock but will rather use a timeout. If the transaction takes more than x seconds, the TM decides that there is a deadlock and the transaction is interrupted. The case with distributed transactions is worse because it will often take longer to detect the presumable deadlock situation and, meanwhile, several transactions are blocked. It's also common that more than one transaction will be affected by the timeout and therefore be terminated.

In any case, deadlocks are bad for our health, both with local and distributed transactions. We can't avoid deadlocks completely, but we can make them less likely to appear. Using short transactions is extremely important for reducing the deadlock risk. The shorter the time you hold the locks, the smaller the risk that somebody else acquires the locks in a way that conflicts with yours. The following are other tips I recommend for decreasing the risk of deadlocks.

Taking `UPDLOCK` When Reading Before Writing

Another common reason for a deadlock is that two transactions first acquire shared locks on one and the same row and then both of them try to escalate their locks to exclusive locks. None of the transactions succeeds until one of the transactions is interrupted. The solution is simple—when you know you need an exclusive lock, acquire it immediately instead of starting with a shared lock that you later escalate to an exclusive lock. You can do that with the `UPDLOCK` optimizer hint in SQL Server, as shown in Listing 6.11.

LISTING 6.11 Example of How to Acquire an Exclusive Lock with a SELECT

```
SELECT description
FROM errand (UPDLOCK)
WHERE id = @id
```

Working with Tables in the Same Order for All Transactions

One simple tip is to always work with your tables in the same order in all transactions. In Listing 6.10, you see that one row is inserted into `errand` and then one row is inserted into `action`. That order should be used for all transactions. I usually say that the master table should be used before the detail table. You don't have a choice when it comes to `INSERT`s as in Listing 6.10 because of `FOREIGN KEY` constraints. When this rule doesn't help because there are no relationships between the tables, use alphabetical order for the table names instead.

Unfortunately, `DELETE` has to happen in the opposite order of master and detail, once again because of `FOREIGN KEY` constraints. The problem is easily solved by first taking an `UPDLOCK` on the master row, and then `DELETE` the detail rows followed by a `DELETE` of the master row.

Obscure Declarative Transaction Design Traps

Using COM+ transactions is often thought of as being simple because the system will deal with the transactions for you, deciding when to `COMMIT` and when to `ROLLBACK`. Even so, it's important to understand how COM+ transactions work and how your settings will affect the outcome.

All three traps that follow are taken from situations where my proposed architecture was not used. Even so, I want to point out a few "gotchas" so you don't fall into these traps if you decide to use another architecture.

Example 1: Incorrect Error Trapping

There are several examples of how error trapping might go wrong, some obvious and some not so obvious. The result might be that `SetAbort()` isn't called at all and that the transaction will therefore be `COMMIT`ed. It's probably obvious why the example in Listing 6.12 isn't one you want to have in your code. When the `Throw()` statement executes in Listing 6.12, the method will stop executing (except for one or more possible `Finally` and/or `Catch` blocks on an outer level).

LISTING 6.12 Incorrect Example of Error Trapping

```
Catch e As Exception
    Throw(e)
    ContextUtil.SetAbort()
```

Example 2: Incorrect Use of `NotSupported`

In this example, we will let a root component control a transaction and ask for help from two secondary components for doing subtasks. Assume you have the components listed in Table 6.5.

TABLE 6.5 Example 2: Instances, Components, and Transaction Attributes

Instances and Components	`Transaction` *Attribute*
`aRoot` (instance of component A)	`TransactionOption.Required`
`aSecondary` (instance of component B)	`TransactionOption.Required`
`anotherSecondary` (instance of component C)	`TransactionOption.NotSupported`

Assume that `aRoot` calls `aSecondary` and `aSecondary` `UPDATE`s a specific errand. (Recall the sample application Acme HelpDesk introduced in Chapter 5.) Then `aRoot` calls `anotherSecondary` that `SELECT`s the same errand, or at least `anotherSecondary` tries to. Because component C is marked with `NotSupported`, its instances will not execute in the same transaction that is still going on for `aRoot` and `aSecondary`. Instead, the `SELECT` in component C will be wrapped in an implicit transaction and put in a wait state waiting for the row in the errand table to be released. Now we're stuck in a wait state until the timeout for the transaction helps us. Unfortunately it won't; it will kill us.

The typical solution to this problem would be to use `Supported` or `Disabled` for component C. (`Disabled` only works if C instances can co-locate in the contexts of A instances.) It could also be the case that a redesign needs to take place. Perhaps the `SELECT` isn't really needed because there may be no `UPDATE TRIGGER` for the table.

Example 3: Incorrect Use of `RequiresNew`

The third example is perhaps a bit strange, but is still possible. This time, we also have three instances and components. You can see how they are configured in Table 6.6.

TABLE 6.6 Example 3: Instances, Components, and Transaction Attributes

Instances and Components	*Transaction Attribute*
`aRoot` (instance of component A)	`TransactionOption.Required`
`aSecondary` (instance of component B)	`TransactionOption.Supported`
`anotherSecondary` (instance of component C)	`TransactionOption.RequiresNew`

This time, `aRoot` calls `aSecondary` and `aSecondary` `SELECT`s a specific errand. Because of the information in the `errand`, `aRoot` understands that the `errand` must be updated; therefore, it calls `anotherSecondary` so that it can deal with the `UPDATE`. No matter what happens afterwards in the activity, the `UPDATE` must take place and therefore it is put in a separate transaction. The result is the same as for the second example; `anotherSecondary` grinds to a halt and waits for the timeout.

As a matter of fact, I have never used `RequiresNew` in any of the systems I have built. The only time I have even thought about it was for my error-logging component, but as I said in Chapter 4, "Adding Debugging Support," I use another trick instead for having the error logged in a separate transaction. The solution to this problem must be a redesign, but first we have to ask ourselves whether it is correct at all that C should run in a transaction of its own. The scenario I presented here was, as I said, a bit strange.

Traps Summary

All three examples of traps that I selected and discussed here were problems arising when several components interacted. None of the problems would have existed if all the code had been put in a single method instead. But that is *not* the moral of this story. That would lead to code bloat. Instead, what I'm saying is that you must understand how different settings affect your transactions and be very careful with error trapping. As usual, COM+ transactions don't change that.

Evaluation of Proposals

As in previous chapters, it's time for me to evaluate the proposals I have presented in this chapter against the criteria I established in Chapter 2.

Evaluation of Transaction Technique Proposal

As I said earlier, the transaction technique to favor is to control transactions in the stored procedures—that is, when you only have one RM. It's a very good idea to let the transactions be controlled in stored procedures when you have a slow network between the components and the database. As a matter of fact, this is most often the solution that gives the best performance

and scalability. When version 1 of .NET has been released, you will find my results of a test in which stored procedure controlled transactions are compared to ADO.NET controlled transactions and COM+ controlled transactions at the book's Web site at `www.samspublishing.com`.

Productivity might be better if you go for automatic transactions instead, because you don't have to code the transactions yourself. In reality, I find the difference in productivity between automatic transactions and local transactions to be small.

Maintainability, reusability, and interoperability may be negatively affected if you don't also prepare for moving to automatic transactions when you need to. Therefore, you should also consider using the proposal I evaluate in the next section.

Evaluation of Transactions in the Architecture Proposal

My proposal of how to deal with transactions in the new architecture is totally independent of the type of consumer, at least as long as you let the consumer stay out of the transaction, which is definitely how it should be. The exception to this is when the consumer is a component that uses automatic transactions, but that is more a matter of interoperability. You should note that the transaction design in the architecture has interoperability as one of its major design goals.

When using pure T-SQL transactions, the most efficient solution for transactions is implemented. This is especially apparent when there is a slow network between the application server and the database server.

Both the performance and the scalability factors are targeted well by the proposal. The main problem is that methods at the Application layer classes that don't need transactions may also have automatic transactions started. There is an easy solution to this (splitting the classes into two parts), but it is not good for maintainability and productivity.

Apart from the drawback just mentioned, I think that the architecture proposal is good for maintainability, reusability, debuggability, and interoperability. To a large extent, this is because of the data access pattern that will be discussed in Chapter 8.

By getting shorter transactions, which is one of the results of the architecture proposal, the reliability will also increase because the risk for deadlocks will decrease.

Evaluation of the Flexible Transaction Design Proposal

The main reason for the flexible transaction design is to increase maintainability. Because of that, reusability and interoperability are improved as well. In the short run, it might be the case that productivity is decreased, but in the long run, it wins.

If you want your components to work well both with automatic and manual transactions, you have to test both situations thoroughly. The proposed design will increase testability because it is easy to just change the settings for some attributes and test again.

What's Next

There has been a lot of talk about business rules in the last few years, but relatively few concrete recommendations for how to handle them have been shown. In the next chapter, I will discuss several different proposals for how to take care of business rules, both in serviced components and in stored procedures.

References

1. T. Ewald. *Transactional COM+: Building Scalable Applications*. Addison-Wesley; 2001.
2. T. Pattison. *Programming Distributed Applications with COM+ and Visual Basic 6.0*. Microsoft Press; 2000.
3. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann; 1993.
4. P. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann; 1997.
5. G. Brill. *Applying COM+*. New Riders; 2000.
6. D. Platt. *Understanding COM+*. Microsoft Press; 1999.