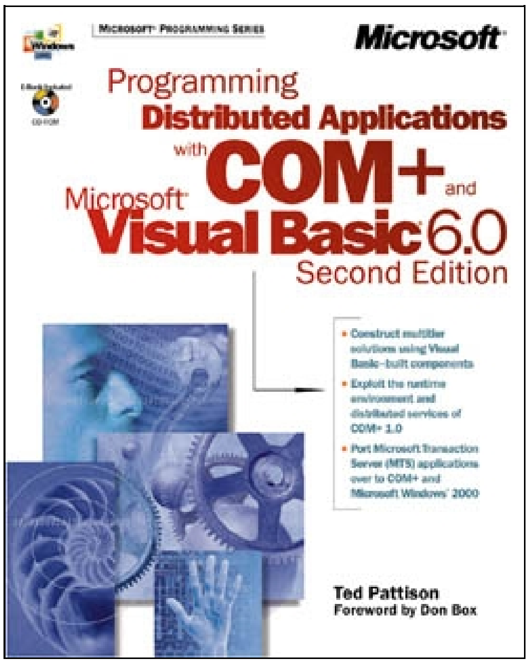
[Next]



[Previous] [Next]

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
Pattison, Ted, 1962-
Programming Distributed Applications with COM+ and Microsoft Visual Basic / Ted Pattison.--2nd ed.
p. cm.
Includes index.
ISBN 0-7356-1010-X
1. Electronic data processing--Distributed processing. 2. Application software--Development. 3. COM (Computer architecture) 4. Microsoft Visual BASIC.
I. Title
QA76.9.D5 P38 2000
004'.36--dc21 00-020234

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QMQM 5 4 3 2 1 0

Distributed in Canada by Penguin Books Canada Limited.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at mspress.microsoft.com. Send comments to mspinput@microsoft.com.

Acquisitions Editor: Ben Ryan
Project Editor: Kathleen Atkins
Technical Editor: Steve Perry

[Previous] [Next]

To my beautiful wife, Amy, and our lovely daughter, Sophie.
I appreciate the sacrifices you make for my writing.
Our love and happiness is what makes everything in life worthwhile.

Special thanks to my parents, who have always given me every opportunity
to be both happy and successful.

[Previous] [Next]

Foreword

A guy walks into a bar and sees two well-dressed women in their mid-fifties talking to one another just within earshot. He hears one of the women saying, "What's your IQ? Mine is 155!" The other woman replies, "Really, mine is 149! How did you interpret the impact of Nelson's micro-economic theory on the development of post-Soviet Union Eastern Europe?" At this point, two men in athletic wear enter the bar in midconversation, catching the ear of our observer. The first man says, "I just had my IQ tested. I am a solid 115!" to which the second replies, "Excellent! My IQ was 108 last time I was tested. Say, who do you like for this Sunday's Giants/Dodgers game?" Not being a sports fanatic, the attention of our eavesdropper turns to two men in their late twenties sitting in a corner booth. He overhears the first man confess, "My IQ test results just came in. I only scored 63." Trying to console his friend, the second man replies, "Don't feel bad, my IQ is only 59. By the way, which is better:

C++ or Visual Basic?"

VB is dead. Finally.

OK, so VB the tool is still alive. But VB as a derogatory label, a culture, an all-encompassing worldview, is dead as a doornail.

For me, the beginning of the end was the first edition of the book you are now reading. Prior to that point, VB was considered an absolute toy amongst most of my colleagues. Worse, VB users were considered to be the people not competent enough to get jobs in documentation, testing, or marketing. VB was often viewed as society's "safety net" that provided "workfare" for those who could not make it on their own as software developers.

And then came Ted's first book.

When developers asked me about good MTS programming books, I constantly found myself recommending Ted's book. At first, I apologized for the VB-related content in the book. "Just pencil in the semicolons if you need them," I would exclaim. However, I eventually grew tired of playing the role of "VB apologist" and decided to rethink my views. At that point, I came to the following realization:

We all are VB programmers in one way or another.

VB programmers are (correctly) portrayed as being blind to a set of issues many consider to be critical. If this is the litmus test for being a VB programmer, then Dim me As VBProgrammer. I care deeply about component integration, yet I am blind to T-SQL optimizations in SQL Server 7.0. My office mate just wrote a 400-page treatise on NT security, yet he couldn't tell you the difference between an unparsed general entity and an internal parameter entity if his life depended on it. In the information-overload world we live in as software developers, selective ignorance is the only way to survive.

Ted (unlike many writers, including myself) has embraced the selective ignorance chromosome that is present in all developers, not just those who have embraced the development tool that dares not speak its name. The first edition of this book demonstrated Ted's mastery of focusing the attention of the reader on the concepts that are important. Unlike the 800-page behemoths so common in this field, Ted's book was a digestible morsel of information that made the able developer more able to solve the problems at hand. This second edition brings Ted's story up to date with the current state of the practice in COM development and will be a valuable addition to the component software canon.

Don Box
Redondo Beach, California

[Previous] [Next]

Acknowledgments

First and foremost, I must acknowledge DevelopMentor. For any of you who are unfamiliar with this company, DevelopMentor focuses on training and education for developers building distributed systems. DevelopMentor is made up of the brightest and most enthusiastic group of computer scientists I have ever encountered—a peer group of the highest level of technical integrity. I consider myself extremely fortunate to have been part of this community for the last six years of my life. Many of the ideas, analogies, and recommendations in this book derive from a shared viewpoint and common base of knowledge that have grown out of the DevelopMentor community.

Many thanks to Mike Abercrombie, Lorrie Trussell, and everyone else who works at DevelopMentor. I am grateful for all your hard work building this rich and nourishing environment, which has given me so many opportunities. DevelopMentor has allowed me to exchange ideas with some of the industry's top researchers and to pursue my passion for cutting-edge technology.

I'd like to give a very special thanks to Don Box. I am especially indebted to Don for his guidance and advice on so many aspects of my career. His vision has been incredibly valuable in structuring the story this book tells. If it weren't for the questionable choices Don makes time after time when buying a laptop computer, I would consider him the smartest man in the software industry today.

I'd like to thank those subject experts who reviewed chapters and provided valuable feedback and constructive criticism. You have made this book considerably better and more accurate. I had lots of support from the different product groups within Microsoft. Thanks to Robert Green from the Visual Studio team for reviewing the first five chapters. Thanks to George Reilly from the IIS teams for reviewing the chapter on IIS and ASP. Thanks to Dick Dievendorff from the MSMQ team for reviewing my messaging chapter. A special thanks to Joe Long from the COM+ team for taking the time to answer countless questions I had when writing the chapters on the internals of COM+.

There were also many people at DevelopMentor who conducted valuable chapter reviews. Thanks to Bob Beauchemin and Jason Masterman for contributing their expertise in the areas of transaction processing and COM+ distributed services. Thanks to Tim Ewald for making excellent recommendations about architectural design issues and explaining how things work way down under. Thanks to Doug Tenure for giving me feedback on some of the earlier chapters on COM. Thanks to Keith Brown, Steve Rodgers, and Dan Sinclair for thoroughly reviewing my security chapter. Thanks to Aaron Skonnard for reviewing several chapters and making excellent suggestions about how to motivate and explain software development based on IIS, ASP, and XML. Thanks to my buddy Brian Randell for his research and his passionate views on the subject of component versioning. Chapter 5 is an adapted version of an article I coauthored with Brian for *Microsoft Systems Journal*. Don't tell anyone (especially Brian), but I think a few of the paragraphs in this chapter might even have been written by him.

DevelopMentor is a great place to work because it's full of legendary characters. Mike Woodring has rewritten the Island Hopper samples using nothing but assembly language. Calvin Caldwell once paused between steps in the ATL COM AppWizard to go out back (on his ranch) and deliver a baby calf. During an intensive week-long course, Dr. Joe Hummel led a cab ride of eager students into Hollywood in the early morning hours to provide a realistic simulation of what it's like to transmit packets across the Internet. And then there's this guy, Fred Wesley, who thinks so hard and so deep that none of us really understands a word he says. The joke about Fred is that he spends so much time contemplating and debating the theory of thought that he never gets around to the practice of thinking.

I also need to thank all my other peers at DevelopMentor including Niels Berglund, Henk de Koning, Jon Flanders, Martin Gudgin, Stuart Halloway, Justin Hoagland, Simon Horrell, Kevin Jones, Paul Kirby, John Lam, Brian Meso, Jose Mojica, Gus Molina, Brad E. Needham, Fritz Onion, Simon Perkins, Brent Rector, Dave Schmitt, George Shepherd, David Smallberg, Dan Weston, and Jason Whittington. All of you have contributed in one way or another to my overall understanding of computer science and software development.

Last but surely not least, I must give recognition to Chris Sells, the evil genius of DevelopMentor. Chris, more than any other individual, is responsible for my career direction. He's the one who ultimately led me down the path to writing about Visual Basic and COM. It was Chris's overly adequate explanations of casting away *const*, template specialization, and functor objects that convinced me that I would never find happiness as a C ++ programmer.

I'd like to thank everyone I have worked with at Fawcette Technical Publications. Thanks to all the hardworking people who put the VBITs conferences together, including Tena Carter, Janet Nickels, Robert

Scoble, Maryam Ghaemmaghami, Jennifer Brucker, and Diamond Jim Fawcette.

Thanks to Joshua Trupin and Joseph Flanigen for publishing my articles in *MSDN Magazine, Microsoft Systems Journal*, and *MIND*. I'd also like to thank the other people at these publications who helped me get my words into print, including Joanne Steinhart, Terry Dorsey, Etna Novik, Joan Levinson, and Michael Longacre.

I would like to thank Claudette Moore and Debbie McKenna at the Moore Literary Agency for their support and assistance. Thanks especially to Claire Horne for helping me create and shop the proposal for the first edition of this book.

I am especially appreciative of all the people at Microsoft Press for helping me put this book together. This includes my acquisitions editor, Ben Ryan, and a very talented editing and production team. Many thanks to Kathleen Atkins for doing an excellent job as lead editor and project manager. Thanks, too, to Sally Stickney for her contributions editing the manuscript. I was also very lucky to have Ina Chang as the manuscript editor and Steve Perry as the technical editor. I hope to work with all of them again in the future.

Finally, I would like to thank the attorneys who work for Microsoft. In retrospect, I am very grateful that you made me spell out "Visual Basic" at every possible occasion instead of allowing me to use the friendly two-letter abbreviation that we all use when referring to our beloved development tool. You have undoubtedly added at least 10 pages to this text and you have, therefore, increased the perceived value of my book to the casual observer looking through the shelves at her local bookstore.

[Previous] [Next]

Introduction

Any developer who wants to create multitier applications using the Microsoft Windows platform as the foundation must rely on many separate pieces of software. Some of these pieces will be supplied by Microsoft, and you or your company will write other pieces. Still other pieces can come from third-party software vendors.

The Component Object Model (COM) is the primary glue that binds all these pieces of software together. It enables programmers and companies to distribute and reuse their code efficiently. COM+, an extended version of COM, provides a valuable runtime environment for components that run in the middle tier. COM+ and a handful of other critical Windows services, such as Internet Information Services (IIS), Active Server Pages (ASP), and Microsoft Message Queuing (MSMQ), provide the building blocks that enable programmers and companies to create large-scale multitier applications.

[Previous] [Next]

Who Is This Book For?

I believe that a thorough knowledge of COM and COM+ is a prerequisite to building multitier applications with Windows 2000. From my perspective, there's no way you can build scalable applications without a solid understanding of the infrastructure that supports the applications. To that end, the purpose of this book is to explain the critical parts of COM+ and Windows 2000 that affect the way you design and write components for a distributed application.

This book is for intermediate and advanced Visual Basic programmers who want to develop for COM+ and Windows 2000. The book focuses on the architecture of the Windows platform. To this extent, it can also serve as a resource for ASP and C++ developers. Some readers might be new to COM. Other readers might already have experience with COM and Microsoft Transaction Server (MTS). My goal in this edition is to accommodate both kinds of readers.

Over the past five years, I've thought long and hard about how to craft a story that includes just the right amount of detail. Some technical details are critical. Other technical details are esoteric, and absorbing them would waste your mental bandwidth. I've tried to cover what's important and omit what's not. I wanted to cover a lot of territory while keeping the text as concise as possible. While I make a point of avoiding unnecessary complexity, I also reserve the right to dive into low-level details in places where I think it's appropriate.

Many Visual Basic programmers don't have the motivation or aptitude to learn about COM and COM+ at this level. Visual Basic can extend a modest amount of the platform's functionality to these programmers without their needing any knowledge of the underlying technology, but they won't be able to create large information systems with COM+, IIS, ASP, and MSMQ. This book is most definitely not for them.

For Readers Already Familiar with the First Edition

I've always hated buying the second edition of a technical book just to find out that it had a new cover and the same old text. One of my main goals with the second edition has been to create something that adds value for those who have read the first edition. Over 75 percent of the text for this book is newly written for the second edition.

I have restructured my coverage of the fundamentals of classic COM and placed them as early in the book as possible. Chapter 2, which covers interface-based programming, is the one chapter that's basically the same in both editions. I've condensed the fundamentals of COM that were spread across several chapters in the first edition into Chapter 3 of this edition. Chapter 4 and Chapter 5 describe using Visual Basic to create and version components. The material in these chapters has been enhanced from the first edition with new coverage of building custom type libraries with IDL and designing components for scripting clients. For programmers who are already comfortable with COM and MTS, Chapters 2 through 5 can serve as a quick review or as a reference.

Chapter 6 and Chapter 7 cover the architecture of the COM+ runtime environment. The aim is to teach you how to write configured components that take advantage of the runtime services and thread pooling offered by COM+. Chapter 8 on transactions introduces new material that compares local transactions to distributed transactions. Chapter 9 includes new and essential coverage of IIS and ASP. Chapter 10 on messaging has coverage of MSMQ similar to the first edition, but it also adds new material about the Queued Components Service and COM+ Events. Chapter 11 is a security chapter recently written from the ground up. (I hope it isn't as painful for you to read as it was for me to write.) Chapter 12 covers application design issues that affect scalability and performance.

What Experience Do You Need?

I assume that you have a background that includes object-oriented programming and creating applications that use classes. It doesn't matter whether you learned about classes using Visual Basic, C++, or Java. It's just important that you understand why you would design a class using encapsulation and that you understand the relationship between a class and an object.

It's helpful but not essential that you have some experience in computer science or a low-level language such as C. It would be impossible for me to tell you about COM without talking about things such as pointers, memory addresses, stack frames, and threads. If you don't have this background, please take the time to contemplate what's going on at a lower level. Occasionally your temples might begin to throb. But the time and effort you invest will be more than worthwhile.

Most readers of this book will have done some work in database programming. It's hard to imagine that an intermediate Visual Basic programmer could have gotten by without having worked on at least one database-oriented application. When I describe writing transactions for COM+ objects, I assume that you have a moderate comfort level with ActiveX Data Objects (ADO) and Structured Query Language (SQL). If you don't have this background, you should acquire it on your own. The ADO and SQL code presented in this book isn't overly complicated.

In the chapter on IIS and ASP, I assume you know the basics of using HTML and creating simple ASP pages. Given the background of most readers and the availability of quality reference material on ASP, I didn't see much point in covering those details.

What's Not in This Book

This book doesn't contain many step-by-step instructions. Therefore, this book won't appeal to those who just want to know what to do but don't care why. Throughout the book, I refer you to MSDN to get the details concerning such practices as using the administrative tools for COM+ or IIS. I don't think these are areas in which most programmers need assistance. My goal is to build your understanding of the theory behind the software.

I don't tell you much about how to automate COM+ or IIS administration through scripting or custom administrative applications. While both COM+ and IIS provide rich object models for automating administrative chores, I don't spend much time on it. I cover a little bit of these topics here and there, but I expect you to learn the details through the documentation in MSDN and the Platform SDK.

If you're looking for a book with a great big sample application that you can use as a starting point, this isn't the right book for you. Most of my code listings are short, between 5 and 10 lines. When I present a code listing, I always try to do it in as few lines as possible to focus your attention on a particular point. I omit extraneous things such as error handling. For this reason, my style doesn't lend itself to those who are looking to blindly copy-and-paste my samples into production code. When it comes to presenting code, my goal is to teach you to fish as opposed to simply giving you fish.

[Previous] [Next]

What's on the CD?

The CD included with this book contains the source code for several Visual Basic applications. These applications contain working examples of the code I use throughout this book. As a Visual Basic programmer, I've always felt that my understanding was boosted by hitting the F5 key and single-stepping through an application line by line. I'd like you to have the same opportunity. The Samples.htm file will give you a synopsis of all the applications and what they do.

The Setup directory on the CD contains the files and instructions for setting up the SQL Server database that you need to run the Market application. The Setup.htm file can walk you through the steps. Each of the other applications that have additional setup instructions will have a Setup.htm file in its directory. I hope you find these sample applications valuable.

What Software Do You Need to Run the Sample Applications?

If you want to run all the sample applications I have included on the CD, you must install Windows 2000 Server or Windows 2000 Advanced Server. Either version includes COM+ and IIS as part of the default installation. If you want to run the messaging example from Chapter 10, you must also install MSMQ, which isn't part of the default installation for Windows 2000.

You should also install Visual Studio 6 including Visual Basic 6, Visual C++, and the most recent version of MSDN. When you install Visual Studio, be sure to select the option to *Register Environment Variables* for reasons that will become clear in Chapter 5. After installing Visual Studio, make sure to install Visual Studio Service Pack 3 or later. To run the sample applications from the later chapters, you must also install SQL Server 7. Once you install SQL Server it never hurts to apply the latest service packs. The final thing to install is the Microsoft Platform SDK. Use the edition from January 2000 or later. The Platform SDK provides many important tools and utilities that will be described throughout the book.

[Previous] [Next]

Updates and Other Information

In my work as a researcher, an instructor, and a writer, I'm continually improving and creating new Visual Basic applications that relate to COM and COM+ programming. You're free to download the most recent collection of samples from my Web site, *http://SubliminalSystems.com*. At this site, you'll also find other information relating to this book, including a listing of any bugs and inaccuracies. If you'd like to send me feedback, mail it to me at *tedp@SubliminalSystems.com*. I hope you enjoy this book.

[Previous] [Next]

Chapter 1

An Overview of COM+

So, one millennium's ended and another's just begun. Just how much impact did the changeover have on your life as a professional developer? Sure, when the clocks all ticked zero, there were a few hiccups here and there. A few outdated systems had to be taken to their final resting places. But our industry as a whole didn't experience the doomsday scenarios predicted in the media. This was especially true for those of us who build applications based on operating systems and development tools created within the last decade. It's pretty humorous when you look back on it all. Rumors of the death of your company's information system were obviously greatly exaggerated.

Then again, does the press really ever know what's going on in our industry? They missed the fact that it was a century bug and not really a millennium bug. The problem was rooted in the use of two-character dates instead of four-character dates. If the human race had had the same technology 100 years earlier, we would have faced all the same issues on New Year's Eve of 1899. However, "millennium bug" sounded so much more sensational and, consequently, sold more newspapers and magazines.

Remember a few years back when the press predicted that everything would be rewritten in Java, including the United States Constitution and the Magna Carta? Now the Java buzz has subsided and we see Java for what it is—a young and promising language with its share of strengths and weaknesses. It didn't take the industry by storm. It's a language that's usable in some situations but unsuitable in others.

Now everybody's talking about XML as the cure-all technology. I know that as XML matures, we'll find more and more uses for it. It will solve lots of problems that are very hard and very real. But XML will never replace technologies such as COM, Corba, and Java, as many people have suggested. It's important that we keep all these new technologies in perspective. It's not healthy to get overly excited about a technology that might solve all your problems a few years down the road when you have to ship an application in the next few months.

[Previous] [Next]

Why Should You Use COM+?

I'll assume that you are at least considering using COM+ to build distributed applications. Many companies are using COM+ and Microsoft Windows 2000 because they provide a robust development platform. This platform is made up of several core technologies that provide the basic building blocks for constructing multitier business applications. And after all, the more the underlying platform brings to the table, the less code you have to write and debug.

I probably don't need to convince you that multitier applications offer many advantages over the two-tier applications that were in vogue in the late 1980s and early 1990s. Your company has probably already decided to abandon the two-tier approach in favor of a multitier strategy. However, I'd like to take a little time at the beginning of this chapter to review the most significant problems with two-tier applications and explain how multitier applications provide solutions to many of these problems. I'll also discuss why multitier development introduces some new problems and additional complexities.

One of Microsoft's goals in developing COM+ has been to offer companies the benefits of multitier applications while hiding as much of the inherent complexity as possible. Over the last decade, Microsoft has made many advances in creating this infrastructure for distributed applications.

The first version of COM shipped in 1993. Since that time, COM has grown from a young and complicated technology to become the core of Microsoft's multitier strategy. This chapter examines COM's most important milestones. Along the way, I'll also do my best to define all the acronyms that marketing folks have generated over the years. You've probably heard of OLE, DCOM, ActiveX, and MTS. COM+ and DNA are the two most recent additions to the list. But have you ever tried to explain the difference between these terms to someone at a cocktail party? It's not easy, is it? They all mean different things to different people.

The chapter concludes with a high-level overview of the distributed services that have been integrated into the COM+ platform. Any nontrivial multitier application requires such things as transaction support, integrated security, a Web server, messaging, and delivery of event notifications. This chapter will identify where each of these COM+ services fits in. Before drilling down into the low-level details in the chapters that follow, I'll show you how all these pieces fit together. This should give you an appreciation for COM+ as a whole and show you the light at the end of the tunnel.

From Two-Tier to Multitier Systems

One of the best reasons to use COM+ is to move a company's information systems from a two-tier architecture to a multitier architecture. This transition requires the design and creation of a middle-tier layer of business objects. Business objects usually sit between client applications and database servers. COM+ serves as the platform for these types of systems.

Two-tier applications have been widely deployed in the industry, so the problems associated with them are well known. Let's review the key shortcomings of a traditional two-tier architecture, such as the one shown in

Figure 1-1.

Figure 1-1 *Typical two-tier applications require a separate connection to a database server for each user, and the user's computer must have the appropriate driver for a specific database management system (DBMS).*

- User interface code is intermingled with business logic and data access code, which makes it difficult to reuse business logic and data access code across multiple client applications. Furthermore, changes to business logic or the database often require rebuilding and redistributing client applications.

- Each client application, complete with business logic and data access code, runs in a separate process, so you can't share resources that are process-specific, such as threads and memory. You also can't share a database connection in a two-tier application because each user requires a separate connection to the database server. The same is true for client applications that connect to a mainframe computer. This inability to effectively share resources across a set of users limits a system's overall throughput and scalability.

- Every client computer requires one or more proprietary drivers so that it can talk to a database server or a mainframe application, which makes desktop computers more expensive to maintain and configure. The system administrator must install and maintain a set of drivers on each computer to provide access to such things as open database connectivity (ODBC) databases. If you need to swap out your back-end database—for example, if you're moving from Sybase to Oracle—the administrator must visit every single desktop. That gets expensive.

- Client applications have a hard time accessing data from multiple data sources. In a corporate environment, it's not uncommon for critical business data to be spread across many different systems, as shown in Figure 1-2. Things get pretty tricky when an application needs to run business transactions using data that's spread across multiple computers. Things get exceptionally difficult when the data is also stored in a variety of formats.

- Client applications are often required to run on the same local area network (LAN) as the database server or mainframe application, which makes it impossible to build distributed applications that span geographic locations. Many companies have employees all over the planet, and many of them need to create applications for customers and suppliers that could never be integrated into such a controlled environment as a corporate LAN.

- Users are restricted to specific platforms. For example, a two-tier system often requires everybody to run the same operating system, such as a 32-bit version of Windows. In a world where any computer can potentially be connected with any other, this limitation is becoming more and more unacceptable.

- Two-tier systems cease to be operational when computers become disconnected, such as when the database server is taken off line for maintenance or laptop users disconnect from the network while they're working in airplanes or at customer sites. A system's availability is severely limited if it experiences downtime whenever various computers can't directly connect to one another.

Figure 1-2 *A two-tier strategy doesn't work well when a company has multiple client applications or multiple database servers.*

Splitting out the presentation tier

The problems of a two-tier architecture can be solved as many other problems in computer science are solved—by adding a layer of indirection. You can decouple client applications from business logic and data access code by introducing a set of *business objects,* as shown in Figure 1-3. The client application containing the user interface code is often referred to as the *presentation tier*. Unless I indicate otherwise, the term *client application* in this book means a presentation-tier application that contains the elements and code for the user interface.

Business objects allow you to centralize your logic and reuse it across several client applications. In networks based on Windows 2000 and Windows NT, these business objects can be deployed using COM as the foundation. COM can also provide the basis for remote communication between client applications and middle-tier objects.

One of COM's biggest selling points is that it allows middle-tier programmers to update the code in their business objects without requiring recompilation or redistribution of client applications. It's pretty straightforward and painless to change your business logic or data access code after your components have already been put into production because client applications are shielded from your business objects' implementation details.

Figure 1-3 *Introducing a set of business objects in the middle tier eliminates costly dependencies between client applications and database servers.*

An application's data can change storage formats and new database servers can be brought on line without undue pain and frustration. You can often make the necessary modifications to a single Microsoft Visual Basic dynamic-link library (DLL). It's easy to recompile a DLL and replace it on a production server in the middle tier. The client applications remain in production unaltered.

In a typical multitier application like the one shown in Figure 1-3, a set of middle-tier objects runs on behalf of many users. However, all these objects often run inside a single process on a Windows 2000 server. This makes it possible to share process-specific resources such as threads, memory, and database connections across multiple users. Also, the computers running client applications don't require database drivers, as they do in the two-tier model.

Logical tiers vs. physical deployment

One of the first questions that arises in multitier development is where each of the tiers should be deployed. Figure 1-4 shows two possible scenarios. The number of tiers doesn't necessarily indicate how many computers are involved. In a small deployment, the business code and the database server might run on the same computer. In a larger system, the data can be kept on one or more dedicated computers while the business objects run on a separate host.

Figure 1-4 *The mapping between business objects and the data access layer can be simple at first and then become more complex without affecting client applications.*

Some physical deployment schemes are easier to set up and less expensive. Others offer higher levels of scalability, reliability, and fault tolerance. When people speak of multitier systems, they're speaking of several distinct *logical tiers*. The physical deployment can vary and can easily be changed after a system is put into production. One of COM's greatest strengths is that it allows you to make changes to the physical deployment without modifying or recompiling any application code.

Some people use the terms *three-tier* or N-*tier* instead of *multitier*. For the most part, these terms all mean the same thing—that a system has three or more tiers. Figure 1-5 shows a more complex set of physical layers. In the multitier model, the business and data access tiers can be arbitrarily complex. The best thing about this model is that client applications know only about a visible layer of business objects. All the additional complexity behind the business objects doesn't concern them. The primary design goal in a multitier system, therefore, is to hide as much of this complexity as possible from the client applications that make up the presentation layer.

Another powerful feature of COM is that it allows client applications to create and use objects from across the network. Behind the scenes, COM uses a protocol called Remote Procedure Call (RPC) to execute method calls across process and host boundaries. RPC is the first of several important protocols that you should understand before you design a distributed application with COM+.

If you have a set of users that all run COM-aware operating systems such as Windows 2000, Windows NT, and Windows 98, you can create multitier applications like the one shown in Figure 1-5. The computers running the client applications rely on COM to establish connections with business objects across the LAN. Once a client application creates business objects from across the network, it can use them to run transactions and retrieve information.

Figure 1-5 *One of the greatest strengths of a multitier architecture is that it hides the complexity of a company's evolving IT infrastructure from client applications.*

Although this approach works well for some applications, it's inadequate for others. This style of development usually requires that every user run a 32-bit version of Windows. It also works much more reliably when all your client applications and business objects are running inside the same LAN. You might, however, want to reach users who haven't logged on to the local network or who are running other operating systems, such as Macintosh, OS/2, or UNIX. You can reach a much larger audience by using a Web-based development strategy.

Web-based applications

The popularity of the Internet and Web-based applications has spurred the industry's adoption of multitier architectures. In a Web-based system, client applications run inside a browser. Browsers submit requests to Web servers using a lightweight protocol called Hypertext Transfer Protocol (HTTP). The presentation tier in a Web-based application is constructed with Hypertext Markup Language (HTML).

The way things work in a typical Web application is actually pretty simple. A client submits a request to the Web server, and the Web server responds by processing the request and sending an HTML-based page back to the user. The great thing about working with HTTP and HTML is that every major platform supports them. Your applications can potentially reach any user on the Internet.

Note that the Web server is the client's entry point into the middle tier, as shown in Figure 1-6. When a client submits an HTTP request, the middle-tier application must load and run business objects in order to realize the benefits of multitier development. While the code and HTML pages that make up the presentation tier live on the server, you can still separate the user interface elements, the business logic, and the data access code. As you'll see in later chapters, you can create a single set of business objects and share them across LAN-based clients as well as Web-based clients.

Figure 1-6 *Web-based development makes it possible to reach clients over the Internet who run a wide range of browsers and operating systems.*

If you design a Web-based application correctly, you can realize all the benefits of multitier development that I've described so far. You can share process-specific resources such as threads, memory, and database connections across a large group of users. You can respond to a single HTTP request by retrieving data and running transactions against multiple data sources. Moreover, Web-based development significantly reduces or eliminates client-side configuration issues.

When you design a Web-based application, you must decide which browsers and platforms you plan to support. If your primary goal is to reach the broadest possible audience, you should opt for a pure HTML solution. If you want to build a more sophisticated user interface, you generally have to restrict users to one specific browser or a set of modern browsers.

For instance, features such as Dynamic HTML (DHTML) and client-side JavaScript are supported by recent versions of Microsoft Internet Explorer and Netscape Navigator. If you use these features, however, users with older browsers will have trouble using your application. Typically, you must decide whether it's more important to support a wider population of users or to create an application with a more polished user interface. It's a tough decision that you should make early in the design phase.

The need for messaging

You should address one final issue during the initial design phase of a multitier application. Many of the protocols used in distributed applications, such as RPC and HTTP, are synchronous. This means that the application running on the client's computer must wait patiently during each request while the server computer performs the requested task. Also, if the server has a large backlog of requests, a client might be forced to wait for an unacceptable period of time.

Also note that applications built using synchronous protocols depend on every computer involved in the processing of a request being on line at the same time. The server must be on line for the client to make requests, and clients must be on line and issuing requests to make use of the available processing cycles that the server has to offer. If the server goes off line, clients can't issue requests. When the clients are off line, the server sits around with nothing to do. In essence, everybody has to be on line at the same time for the system as a whole to be operational.

Frequently, multiple applications need to communicate in an asynchronous and disconnected manner. *Messaging* is a mechanism for establishing asynchronous communication between two or more applications. Client applications send asynchronous messages to queues across the network. Each message typically represents a client request or some type of notification. Server applications monitor these queues and process messages either as they arrive or at a later time. Messages and queues simply provide a layer of indirection between the applications that are making requests and the applications that are processing requests.

An application built on top of a messaging protocol can continue to operate when various computers are off line. However, creating a robust messaging infrastructure that can route each message to its destination queue with the required delivery guarantees is not a trivial undertaking. A messaging infrastructure must provide an extensive set of subsystems to transparently store messages when computers are off line and then forward them to their destination later when a connection can be established.

Most companies aren't willing to invest the time and money it takes to write and debug the code for a custom messaging infrastructure. Fortunately, several messaging products that offer the benefits of asynchronous and connectionless communication are commercially available, such as the IBM MQSeries family and Microsoft Message Queuing Services (MSMQ). This eliminates the need to hand-roll a custom messaging infrastructure.

[Previous] [Next]

The Evolution of Microsoft's Multitier Platform

As I mentioned earlier, Windows 2000 and COM+ offer a platform for building multitier applications. But many of the core technologies that make up this platform have been around for a long time. It's important that you understand how the various pieces of the platform have evolved over the years.

As far back as the early 1990s, influential individuals at Microsoft realized that a disproportionate amount of money was being spent on infrastructure in the development of large multitier systems. They realized that companies wanted to spend more time writing custom business logic and less time writing complex code to address issues such as sharing middle-tier resources and monitoring distributed transactions. Microsoft's entire multitier strategy is based on the assumption that companies would rather have someone else create the generic yet critical pieces of a framework for distributed applications.

As Microsoft's multitier strategy has evolved, its attempts to give it a name and a consistent identity have created some confusion. The marketing folks seem to dream up entirely new names and acronyms every year or two for technologies that have already been around for a while. For example, Microsoft's latest name for its platform is the Windows Distributed interNet Applications Architecture (Windows DNA). From a marketing perspective, Microsoft needs new fresh names to compete with similar technologies such as Corba and Enterprise JavaBeans. From your perspective as a developer, these name changes don't mean much. Try not to let them confuse you. No one's reinventing the wheel here—they're simply starting to call it a Circular Locomotion Device (CLD).

Every topic discussed in this book fits under the vast umbrella of Windows DNA. I won't even attempt to cover all the DNA-related technologies, such as DHTML and client-side scripting. The term *DNA* encompasses everything that Microsoft has ever done to help companies build multitier applications. Many programmers avoid the term altogether because they don't want to be confused with someone from the marketing department or with a nontechnical manager. If you really want to talk shop in development circles, you have to understand all the important technologies that make up the platform.

The Foundation: COM

Microsoft's multitier strategy is founded on a core technology known as the Component Object Model (COM). While COM offers many benefits, it is a complex technology that involves several challenging concepts and tons of low-level details. Some of these concepts and many of these details are critical to your understanding of how to properly build middle-tier components for a COM+ application. Many other details associated with COM are no longer relevant or are important only to programmers who are building presentation-tier applications. You need not concern yourself with such COM-related topics as object linking and embedding (OLE), ActiveX controls, and connection points (such as Visual Basic events), so I have omitted them from this book. This book focuses exclusively on the details of COM that are important for programmers who are building nonvisual components for the middle tier.

The term *COM* means many different things to many different people. On the one hand, it's a specification for writing reusable software that runs in component-based systems. On the other hand, it's a sophisticated infrastructure that allows clients and objects to communicate across process and computer boundaries. Many developers who are already COM-savvy see it as a new programming style and a set of disciplines that are required in order to work in a Microsoft-centric environment.

The COM programming model is based on the distribution of class code in binary components. This means that software (components) that adheres to COM can be reused without any dependencies on source code. Developers can ship their work as binary files without revealing their proprietary algorithms. The reuse of code in a binary form also eliminates many compile-time problems that occur when applications are assembled using a development style based on source code reuse.

Before component-based technologies such as COM were available, large applications were built by sending hundreds or even thousands of source files to a compiler in a single batch job to build one executable file. This development style, with its reliance on *monolithic applications,* requires huge executables and long build times. Also, to take advantage of a modification to a single line of source code, the entire application must be rebuilt. This makes it increasingly difficult to coordinate the programming teams working together on a large application. Maintaining and enhancing a monolithic application is awkward at best.

Component-based development solves many of the problems associated with monolithic applications. It allows development teams to ship binary files rather than source code. Binary components can be updated independently and replaced in the field, which makes it much easier to maintain and extend an application after it's been put into production. Most people agree that using COM or some other component-based technology is an absolute requirement in the development of a large information system.

COM is based on object-oriented programming (OOP). This means that COM is about clients communicating with objects. COM exploits the OOP paradigm to achieve higher levels of reuse and maintainability than is possible with other models of binary reuse. COM clients and COM classes typically live in separate binary files. COM defines an infrastructure that enables clients to create and bind to objects at runtime.

A platform based on object-oriented component reuse must provide a dynamic class-loading mechanism. This is one way in which Java and COM are alike. A client creates objects at runtime by naming a specific class that's been compiled into a separate binary component. A system-supplied agent tracks down the class code, loads it, and creates the object on behalf of the client.

Binary reuse makes it far easier to incorporate small changes to an application. For example, a minor bug fix or a performance modification can be made to a DLL. The DLL can then be recompiled and replaced in the field without adversely affecting any of the client applications that use it. Systems based on source code reuse must typically recompile every line of code in the entire application, which makes maintaining and extending software cumbersome and expensive.

The principles of binary reuse allow you to construct COM-based applications using language-independent components. When several teams are building components for a single system, each team can choose its programming language independently. Today's list of COM-enabled languages includes C++, Visual Basic, Java, Delphi, and even COBOL. Each team can select a language that matches its programming expertise and gives it the best mix of flexibility, performance, and productivity.

For example, if one team requires low-level systems code, it can use C++ for its flexibility. Another team that's writing and extending business logic and data access code for the same application can use Visual Basic for its high levels of productivity. The ability to mix and match languages makes it easier for companies to make the best use of their existing pools of programming talent.

Interface-based programming

Microsoft engineers made many important architectural decisions as they designed COM. But one of the most profound decisions was to require that COM have a formalized separation of interface from implementation. This means that COM is founded on the idea of *interface-based programming*.

The concept of interface-based programming wasn't a clever new idea from Microsoft engineers. This programming style had already been adopted by academic computer scientists and by organizations that needed to build large, extensible applications. Interface-based programming was pioneered in languages such as C++ and Smalltalk, which have no formal support for the concept of a distinct, stand-alone interface. Today languages and tools such as Java and Visual Basic have built-in support for this style of programming. So while Microsoft doesn't get credit for the idea, it should definitely get credit for seeing the elegance of interface-based programming and using it as the cornerstone of COM.

An interface, like a class, is a distinct data type. It defines a set of public methods without including any implementation. In another sense, an interface defines a very specific protocol for the communication that occurs between a client and an object. The act of decoupling the interface from the class or classes that implement it gives class authors the freedom to do many things that would otherwise be impossible. A developer writing a client application against an interface definition avoids dependencies on class definitions. Chapter 2 introduces core concepts of interface-based programming. While it might be one of the more challenging chapters of this book, its concepts are important to your understanding of how and why COM works the way it does.

Distributed COM

From the beginning, COM was designed to transcend process boundaries. The earlier releases of COM made this possible only when the client process and the server process were running on a single computer. With the release of Microsoft Windows NT 4, support was added to COM that allowed this interprocess communication to extend across computer boundaries. Microsoft created a new wire protocol for COM that made it possible to deploy distributed applications in a LAN environment. This was a significant milestone in Microsoft's strategy for enterprise computing.

At first, Microsoft struggled to come up with a marketing term to signify that COM could finally be used to create objects from across the network. The name Distributed COM (DCOM) won out over Network OLE when Windows NT 4 was first shipped in August 1995. Today neither term is in style among developers. *Distributed COM* is currently the proper term for talking about COM's wire protocol, although many developers think that COM itself is a distributed technology and that putting *Distributed* in front of *COM* is redundant.

As I mentioned earlier, COM's support for distributed applications is based on an interprocess mechanism named Remote Procedure Call (RPC). RPC is an industry standard that has matured on many platforms. Microsoft enhanced RPC with object-oriented extensions to accommodate COM, and the resulting implementation on the Windows platform is known as Object RPC (ORPC).

COM and RPC have a symbiotic relationship. COM offers RPC an object-oriented feel, and RPC offers COM the ability to serve up objects from across the network. In Chapter 3, we'll look at how COM leverages RPC to transparently call methods on remote computers by sending request and response packets between clients and objects and how clients still invoke methods as usual, as if the objects were nearby. It's remarkable how COM's architects were able to hide so many of the details required by RPC from both middle-tier programmers and client-side programmers.

From COM to MTS

The release of Microsoft Transaction Server (MTS) was a significant milestone in the evolution of the platform.MTS is a piece of software created for Windows NT Server. MTS allows business objects running on Windows NT Server to run and control distributed transactions from the middle tier. However, MTS is much more than a transaction monitor; it provided a brand-new runtime environment for COM objects running in the middle tier. MTS added lots of critical infrastructure support that wasn't included with COM. In particular, it added new support for distributed transactions, integrated security, thread pooling, and improved configuration and administration.

The name MTS has caused confusion because the software wasn't created just for companies that want to run distributed transactions. MTS provided a vehicle for Microsoft to ship the next generation of its distributed application framework. Middle-tier objects targeted for Windows NT Server should be run in the MTS environment whether or not they're involved in transactions. People get confused when they learn that MTS can be used to deploy objects that are nontransactional. To eliminate this confusion in Windows 2000, Microsoft has changed the name of the middle-tier runtime environment from MTS to COM+. The transaction support that was created for MTS is also included in COM+. If you already know how to program transactions using MTS, you don't need to learn much more to write transactional components for COM+. Things work almost exactly the same way. (Programming transactions with COM+ is covered in Chapter 8.)

In addition to supporting distributed transactions, MTS extended COM's security model. MTS security is based on the notion of *roles*. A role is an abstraction that represents a security profile for one or more users in an MTS application. At design time, a developer can set up security checks using roles in either a declarative or a programmatic fashion. At deployment time, an administrator maps a set of roles to user accounts and group accounts inside a Windows NT domain. The role-based security model of MTS provides more flexibility and more granularity than the original security model provided by COM.

Another significant feature that MTS added to the platform was a scheme to manage concurrency by conducting thread pooling behind the scenes. MTS introduced an abstraction called an *activity,* which represents a logical thread of execution in an MTS application. MTS programmers should write their applications to provide one activity per client application. The MTS runtime automatically binds logical activities to physical threads. Once the number of clients reaches a predefined threshold, the MTS runtime begins sharing individual threads across multiple clients.

Multitier applications that use a thread-pooling scheme scale better than those that use a single-threaded model or a thread-per-client model. A single-threaded model removes any chance of concurrency because it can't execute methods for two or more clients at the same time. A thread-per-client model results in unacceptable resource usage in larger applications because of the ongoing need to create and tear down physical threads. The goal of a thread-pooling scheme such as the one built into MTS is to create an optimized balance between higher levels of concurrency and more efficient resource usage. In this respect, MTS takes on a pretty tough task and neatly tucks it under the covers.

One other significant feature of MTS was improved support for computer configuration and network management. COM made it tricky and expensive to deploy and administer larger COM-based applications in a network environment. The MTS administration tools made it much easier to configure and manage the server computers that run middle-tier objects. Unlike COM, MTS makes it possible to manage many server computers from a single desktop. MTS also provides the tools to generate client-side and server-side setup programs.

Attribute-based programming and interception

MTS introduced a significant concept to the platform's programming model: attribute-based programming. Services provided by the platform are exposed through declarative attributes. Attribute settings are determined by programmers at design time and can be reconfigured by administrators after an application has been put into production. Attribute-based programming is very different from the way that most operating systems have exposed system services in the past.

Traditionally, operating systems have exposed services through a set of functions in a call-level application programming interface (API). In this model, an application leverages system services by making explicit calls to the API. This means that you must compile explicit system-level calls into your application. If you want to change the way you use a system service after an application is already in production, you must modify your code and recompile the application. A programming model based on declarative attributes is much more flexible.

Let's look at an example to give you a clearer picture of how declarative attributes work. A component in an MTS application has an attribute that tells the MTS runtime environment whether it supports transactions. If a programmer marks a component to require a transaction, objects instantiated from the component are created inside the context of a logical MTS transaction. When a client calls one of these transactional objects, the MTS runtime automatically makes a call to start a physical transaction. It also makes the appropriate call to either commit or roll back the transaction. The point is that programmers never make explicit calls to start, commit, or abort a transaction. All the necessary calls are made behind the scenes by the MTS runtime.

Why is the attribute-based programming model better than the older procedural model, which requires explicit API calls? First, the new model requires less code. You indicate your preferences at design time by setting attributes, and the underlying runtime environment makes sure your needs are met. A second, less obvious reason is that administrators can easily reconfigure the way that an application uses a system service after it's been put into production. There's no need to modify any code or recompile your application.

The attribute-based programming model of MTS relies on a mechanism known as *interception*. When a client creates an object from a component that's been configured in an MTS application, the underlying runtime inserts an interception layer, as shown in Figure 1-7. This interception layer represents a hook that allows the MTS runtime to perform system-supplied preprocessing and postprocessing on method calls. The system steals away control right before and right after an object executes the code behind each method.

Figure 1-7 *An attribute-based programming model relies on interception to leverage system services.*

The COM vs. MTS problem

One of the biggest dilemmas for programmers who are building multitier applications targeted for Windows NT Server is deciding when to use COM versus MTS. Many companies have chosen MTS because it offers many valuable built-in services that COM doesn't offer. Other companies have chosen COM because they don't understand the added value of the MTS runtime environment.

COM ships with Windows NT Server, while MTS does not. To use MTS, you must install an extra piece of software, the Windows NT Option Pack. This means that MTS isn't really part of COM. COM has its own programming model and runtime layer; MTS has a different programming model and separate runtime layer.