ATL INTERNALS

SECOND EDITION
WORKING WITH ATL 8

CHRISTOPHER TAVARES
KIRK FERTITTA
BRENT RECTOR
CHRIS SELLS

ATL Internals

Second Edition

The Addison-Wesley Object Technology Series

Grady Booch, Ivar Jacobson, and James Rumbaugh, Series Editors
For more information, check out the series web site at www.awprofessional.com/otseries.

Ahmed/Umrysh, *Developing Enterprise Java Applications with J2EE™ and UML*

Arlow/Neustadt, *Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML*

Arlow/Neustadt, *UML 2 and the Unified Process, Second Edition*

Armour/Miller, *Advanced Use Case Modeling: Software Systems*

Bellin/Simone, *The CRC Card Book*

Bergström/Råberg, *Adopting the Rational Unified Process: Success with the RUP*

Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*

Bittner/Spence, *Managing Iterative Software Development Projects*

Bittner/Spence, *Use Case Modeling*

Booch, *Object Solutions: Managing the Object-Oriented Project*

Booch, *Object-Oriented Analysis and Design with Applications, 2E*

Booch/Bryan, *Software Engineering with ADA, 3E*

Booch/Rumbaugh/Jacobson, *The Unified Modeling Language User Guide, Second Edition*

Box et al., *Effective COM: 50 Ways to Improve Your COM and MTS-based Applications*

Buckley/Pulsipher, *The Art of ClearCase® Deployment*

Carlson, *Modeling XML Applications with UML: Practical e-Business Applications*

Clarke/Baniassad, *Aspect-Oriented Analysis and Design*

Collins, *Designing Object-Oriented User Interfaces*

Conallen, *Building Web Applications with UML, 2E*

Denney, *Succeeding with Use Cases*

D'Souza/Wills, *Objects, Components, and Frameworks with UML: The Catalysis(SM) Approach*

Douglass, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*

Douglass, *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*

Douglass, *Real Time UML, 3E: Advances in The UML for Real-Time Systems*

Eeles et al., *Building J2EE™ Applications with the Rational Unified Process*

Fowler, *Analysis Patterns: Reusable Object Models*

Fowler, *UML Distilled, 3E: A Brief Guide to the Standard Object Modeling Language*

Fowler et al., *Refactoring: Improving the Design of Existing Code*

Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*

Gomaa, *Designing Software Product Lines with UML*

Heinckiens, *Building Scalable Database Applications: Object-Oriented Design, Architectures, and Implementations*

Hofmeister/Nord/Dilip, *Applied Software Architecture*

Jacobson/Booch/Rumbaugh, *The Unified Software Development Process*

Jacobson/Ng, *Aspect-Oriented Software Development with Use Cases*

Jordan, *C++ Object Databases: Programming with the ODMG Standard*

Kleppe/Warmer/Bast, *MDA Explained: The Model Driven Architecture™: Practice and Promise*

Kroll/Kruchten, *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*

Kroll/MacIsaac, *Agility and Discipline Made Easy: Practices from OpenUP and RUP*

Kruchten, *The Rational Unified Process, 3E: An Introduction*

LaLonde, *Discovering Smalltalk*

Lau, *The Art of Objects: Object-Oriented Design and Architecture*

Leffingwell/Widrig, *Managing Software Requirements, 2E: A Use Case Approach*

Manassis, *Practical Software Engineering: Analysis and Design for the .NET Platform*

Marshall, *Enterprise Modeling with UML: Designing Successful Software through Business Analysis*

McGregor/Sykes, *A Practical Guide to Testing Object-Oriented Software*

Mellor/Balcer, *Executable UML: A Foundation for Model-Driven Architecture*

Mellor et al., *MDA Distilled: Principles of Model-Driven Architecture*

Naiburg/Maksimchuk, *UML for Database Design*

Oestereich, *Developing Software with UML, 2E: Object-Oriented Analysis and Design in Practice*

Page-Jones, *Fundamentals of Object-Oriented Design in UML*

Pohl, *Object-Oriented Programming Using C++, 2E*

Quatrani, *Visual Modeling with Rational Rose 2002 and UML*

Rector/Sells, *ATL Internals*

Reed, *Developing Applications with Visual Basic and UML*

Rosenberg/Scott, *Applying Use Case Driven Object Modeling with UML: An Annotated e-Commerce Example*

Rosenberg/Scott, *Use Case Driven Object Modeling with UML: A Practical Approach*

Royce, *Software Project Management: A Unified Framework*

Rumbaugh/Jacobson/Booch, *The Unified Modeling Language Reference Manual*

Schneider/Winters, *Applying Use Cases, 2E: A Practical Guide*

Smith, *IBM Smalltalk*

Smith/Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*

Tavares/Fertitta/Rector/Sells, *ATL Internals, Second Edition*

Tkach/Fang/So, *Visual Modeling Technique*

Unhelkar, *Process Quality Assurance for UML-Based Projects*

Warmer/Kleppe, *The Object Constraint Language, 2E: Getting Your Models Ready for MDA*

White, *Software Configuration Management Strategies and Rational ClearCase®: A Practical Introduction*

The Component Software Series

Clemens Szyperski, Series Editor
For more information, check out the series web site at www.awprofessional.com/csseries.

Cheesman/Daniels, *UML Components: A Simple Process for Specifying Component-Based Software*

Szyperski, *Component Software, 2E: Beyond Object-Oriented Programming*

ATL Internals
Second Edition

Working with ATL 8

Christopher Tavares
Kirk Fertitta
Brent Rector
Chris Sells

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

ATL internals : working with ATL 8 / Chris Tavares... [et al.]. — 2nd ed.
p. cm.
Previous ed. by Brent Rector.
ISBN 0-321-15962-4 (pbk. : alk. paper) 1. Application software—Development. 2. Active template library. I. Tavares, Chris. II. Rector, Brent. ATL internals.

QA76.76.D47R43 2006
005.3—dc22

2006008998

ISBN 0-321-15962-4
Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
First printing, July 2006

Chris Tavares:

For Wendy, who believed in me even when I didn't believe in myself.

Kirk Fertitta:

To Mom and Ed, for their unwavering faith and encouragement.

Brent Rector:

To Lisa, for being there from the start and providing encouragement constantly, for nearly three decades now.
To Carly and Sean, who don't think it's that big of a deal that their dad writes books; it's all they've ever known.

Chris Sells:

This edition is dedicated to the native Windows programmers that .NET has left behind. I hope that .NET will serve your needs in the future, but until then, this book is for you.

This page intentionally left blank

Contents

This page intentionally left blank

Foreword to the Second Edition

Wow. It has been a long time since I wrote the foreword for the first edition of *ATL Internals*. Reading through the old introduction really takes me down memory lane; I can hardly believe that it has been almost eight years. Not long after I wrote it, I moved on to the Windows team at Microsoft and then on out of Microsoft a year later. I came back to Microsoft (and the Visual C++ team) a few years ago, and I am now managing several development teams in Visual C++. One of these is the libraries team, of which ATL is a part, and it is fun to be involved in ATL again. Jan and Christian have both moved on, although Nenad expanded the windowing classes from ATL that I mentioned in the first introduction into a separate library called WTL (Windows Template Library[1]). WTL is now a Microsoft open-source project that Nenad manages.

ATL has changed in ways I never could have predicted, and it has been bittersweet to see it continue to grow without being personally involved. There have been many great people who have worked on ATL over the years. Some of them I have known quite well and others I never knew.

When I mentioned "some new ways of accessing the ATL functionality" in the first foreword, I was referring to attributes. This technology was delivered in Visual Studio .NET 2002, but it never really developed into what we envisioned. ATL attributes still work in the current release and they can be quite powerful, but there are no plans to expand their use. This new version of ATL Internals provides lots of updates and does cover attributes, but doesn't assume that you're going to depend on this feature. This edition also includes a very nice introduction to ATL Server, which provides a flexible, high-performance way to create web applications. If performance is a critical requirement, ATL Server was built for you. Other ATL 8 improvements include better security, full 64-bit support, better scalability, debugging improvements, support for C++/CLI, and managed ATL components.

What has become the .NET ecosystem was just getting underway back in 1998. It has revolutionized programming for many developers and will continue to deliver improvements in the years to come. However, COM programming (and

[1] http://wtl.sourceforge.net

ATL) is still very much alive and is very important to many developers both inside and outside of Microsoft. The second edition of this book, like the first, provides the details you need to maximize your investment in those technologies.

Jim Springfield
April, 2006

Foreword to the First Edition

When I first saw the title of this book, I told Chris Sells that it sounded like the book I always wanted to write. Ever since we released ATL, some of us have been saying, "We should write a book on how ATL works." After reading *ATL Internals*, I don't think there would be much left for me to write about. Actually, this is kind of a relief. At this point, I think most aspects of ATL have been covered, and *ATL Internals* provides an excellent source of information of the inner workings of ATL. So, Chris asked me to provide some information that can't be deduced by looking at the ATL source code.

A Brief History of ATL

I first got into templates in late 1995, while I was a developer on the MFC team. A friend of mine here was evaluating various STL vendors for the Visual C++ product, and he talked to me a lot about templates. I played around with templates a bit, but didn't do much with them. Soon after, the VC team split off an enterprise team to focus solely on Visual C++ 4.2 Enterprise (our first VC enterprise product). I moved over to head up the libraries work for VCEE. At the time, we explored several different ideas. Microsoft Transaction Server was just getting started then, and we talked a lot with them about COM, transactions, databases, and middle-tier business objects. Pretty quickly, we realized that we needed a better mechanism for creating COM objects from C++. Jan Falkin and Christian Beaumont were working for me at that time. Jan was working on an automation interface to ODBC data sources, and Christian was working on a template-based access method to ODBC data sources (a forerunner of our current OLEDB consumer templates). I was working on the COM infrastructure, since everything was already pointing to COM back then.

Initially, I was just playing around with COM and templates, but slowly it started to stabilize around a few important concepts. From the beginning I wanted to support all threading models, but I didn't want to pay for it unless it was needed. The same was true for aggregation. I didn't want anyone to have an excuse not to use ATL. (I didn't want to hear, "Well, ATL is cool, but I can save a couple of bytes if I do this myself.") So, performance and flexibility came before ease of use when

that decision had to be made. One of the main concepts that came out of this was that the class a user writes is not the class that was actually instantiated. This allowed many optimizations that otherwise could not have occurred. Some of the other concepts were multiple inheritance for interfaces, "creator" functions, and a data-driven COM map. We started to show this around and got a lot of good feedback on it. Several people thought we should get this out to customers as soon as possible, so we decided to RTW (release to the web) in the early summer of '96. That was ATL 1.0. Our working name for our libraries was MEC (Microsoft Enterprise Classes), but our marketing person thought we should have something that more reflected what we were doing. Because of our COM focus and the fact that at the time, everything was being called "Active" something or other, we selected the name *Active Template Library*. We got a good reception for it and in late summer of '96 we released ATL 1.1. By this time, Jan and Christian had started working directly on ATL. ATL 1.1 had bug fixes and support for a few more features such as connection points, NT services, RGS registry support, and security.

After ATL 1.1, we started working on ATL 2.0. Its primary focus was the creation of ActiveX controls. Jan and Christian did much of the work on this, while I still focused on the core stuff (such as rewriting the connection points to make them smaller). Nenad Stefanovic also joined us at that time and started work on the windowing support in ATL, as well as doing the composite control support in VC 6.0. We were originally planning on ATL 2.0 to be shipped on the web targeting VC 4.2. However, our plans changed and we changed ATL 2.0 to ship in VC 5.0 (12/96), and shipped ATL 2.1 with the Alpha version of Visual C++ 5.0. The only difference between ATL 2.0 and 2.1 were some bug fixes for Alpha, MIPS, and PowerPC. We also simultaneously shipped ATL 2.1 on the web with AppWizard and ObjectWizard support for VC 4.2. After a couple of months of working on ATL 3.0 (called ATL 2.5 at the time), Christian and I were burned out and took some time off from ATL, while Jan took over as ATL lead. A few months later, we came back, and Christian became the ATL lead while I moved on to explore some other things for Visual C++, although I do still get into the source code every now and then.

We shipped VC 6.0 in June '98 and are currently working on the next release. Expect to see lots of cool new stuff in ATL as well as some new ways of accessing the ATL functionality. I am glad to see ATL continue to evolve, while at the same time maintaining the original goals of generating small, efficient code. So, take a look at this book, learn some new tricks, and gain a deeper understanding of how it all works.

Jim Springfield
October, 1998

Preface

.NET has hit the Windows programmer community like a tornado, tipping over the trailer homes of the ways that we used to do things. It's pretty much swept up the needs of most web applications and service applications, as well of most of the line-of-business applications for which we previously used Visual Basic and MFC.

However, a few stubborn hold-outs in their root cellars will give up their native code only at the end of a gun. These are the folks with years of investment in C++ code who don't trust some new-fangled compiler switches to make their native code "managed." Those folks won't ever move their code, whether there are benefits to be gained or not. This book is partially for them, if they can be talked into moving their ATL 3/Visual C++ 6 projects forward to ATL 8 and Visual Studio 2005.

Another class of developers that inhabit downtown Windows city aren't touched by tornados and barely notice them when they happen. These are the ones shipping applications that have to run fast and well on Windows 95 on up, that don't have the CPU or the memory to run a .NET application or the bandwidth to download the .NET Framework even if they wanted to. These are the ones who also have to squeeze the maximum out of server machines, to take advantage of every resource that's available. These are the ones who don't have the luxury of the CPU, memory or storage resources provided by the clear weather of modern machines needed for garbage collection, just-in-time compilation, or a giant class library filled with things they don't need. These developers value load time, execution speed, and direct access to the platform in rain, sleet, or dark of night. For them, any framework they use must have a strict policy when it comes to zero-overhead for features they don't use, maximum flexibility for customization, and hard-core performance. For these developers, there's ATL 8, the last, best native framework for the Windows platform.

For clients, ATL provides windowing, COM client smart types, extensive COM control and control hosting, MFC integration (including several MFC classes that no longer require the rest of MFC), and web service proxy generation. For servers, ATL provides full COM server and object services, and extensive support for high-throughput, high-concurrency web applications and services. For both clients and services, ATL makes aggressive use of macros and templates to give you maximum flexibility and low overhead, making sure you pay for only the features you use and giving you full transparency via the source code into how those classes map their

functions to the platform. For productivity, ATL provides a full set of wizards for starting and building client and server projects.

Attributes

Pushing the productivity idea, in ATL 7 and Visual Studio 2003, the ATL team introduced attributed ATL, allowing ATL programmers to annotate their code using the same techniques that you would use to add metadata to IDL interfaces and coclasses (such as the `uuid` attribute). In fact, the wizards were so happy to show you this style of code that, in VS03, the Attributed option was on by default. However, all is not sunshine and bluebirds with attributes. In .NET and IDL, attributes are a real part of the programming model; support for them exists all the way down. In ATL, attributes are more of a compiler trick, like super-macros, generating base classes, macro maps, Registry scripts, and IDL files.

Unlike macros, however, ATL attributes are not transparent—you can't see what is going on very well. A compiler switch is included to show a representation of generated code, such as what base classes were added, but it has regressed in VS05. This has led to problems in understanding and debugging issues, which was not helped by bugs in the attribute-generated code. That's not to say that the rest of ATL is bug free (or that any software is bug free), but when it comes to problems in base classes or macros, ATL has always enabled you to replace problem functionality in several ways. In fact, code to work around problems was a big part of the first edition of this book because you could so easily sidestep problems.

Cues in VS05 indicate that attributes are no longer a major part of the ATL team's focus. For example, the compiler switch shows less information, not more, about what attributes generate. Most telling, however, is that the Attributed option in the VS05 wizards is no longer checked by default.[1] For that reason, although we cover the principles of ATL attributes in Appendix D, "Attributed ATL," you won't find them sprinkled throughout the book. We believe that half-hearted attributes won't make ATL 8 programmers the happiest with their native framework of choice.

Audience

This book is for the C++/COM programmer moving to ATL 8, as provided with Visual Studio 2005. ATL was built with a set of assumptions, so to be an effective ATL programmer, you need to understand not only how ATL is built, but also why.

[1] Except for when generating an ATL Server Web Service project, when the Attributed Code option is checked and disabled so that you can't uncheck it.

Of course, to understand the why of ATL, you must understand the environment in which ATL was developed: COM. Instead of attempting to compress all required COM knowledge into one or two chapters, this book assumes that you already know COM and spends all its time showing you the design, use, and internals of ATL. Don Box's *Essential COM* (Addison-Wesley Professional, 1997) is a good source of COM knowledge, if you'd like to brush up before diving into ATL.

Outline

With the exception of the first chapter, this book is arranged from the lowest levels of ATL to the highest; each chapter builds on knowledge in previous chapters. The first chapter is a brief overview of some of the more common uses for ATL and the wizards that aid in those uses. Whenever things get too detailed in the first chapter, however, we refer you to a subsequent chapter that provides more in depth coverage.

Chapters 2 through 5 present the core of ATL. Chapter 2, "Strings and Text," covers the messy world of string handling in C++, COM, and ATL. Chapter 3, "ATL Smart Types," discusses ATL smart types, such as `CComPtr`, `CComQIPtr`, `CComBSTR`, and `CComVariant`. Chapter 4, "Objects in ATL," discusses how objects are implemented in ATL and concentrates on the great range of choices you have when implementing `IUnknown`. Chapter 5, "COM Servers," discusses the glue code required to expose COM objects from COM servers. Chapter 6, "Interface Maps," delves into the implementation of `IUnknown` again, this time concentrating on how to implement `QueryInterface`; this chapter shows techniques such as tear-off interfaces and aggregation. Chapters 7, "Persistence in ATL"; 8, "Collections and Enumerators"; and 9, "Connection Points," discuss canned interface implementations that ATL provides to support object persistence, COM collections and enumerators, and connection points, respectively. These services can be used by components that might or might not provide a user interface. Chapters 10, "Windowing"; 11, "ActiveX Controls"; and 12, "Control Containment," concentrate on building both standalone applications and user interface components. These chapters cover the ATL window classes, controls, and control containment, respectively. Finally, Chapters 13, "Hello, ATL Server," and 14, "ATL Server Internals," cover ATL Server, which lets you build web applications on IIS. Chapter 13 introduces ISAPI and ATL Server, and Chapter 14 looks under the hood of ATL Server.

Much of what makes the ATL source difficult to read is its advanced use of templates. Appendix A, "C++ Templates by Example," provides a set of examples that illustrate how templates are used and built. If you've seen ATL source code and wondered why you pass the name of a deriving class to a base class template, you will find this appendix useful. Appendix B, "ATL Header Files," provides a list of the ATL header files, along with descriptions to help you track down your favorite parts of the ATL implementation. If you're already an ATL 3 programmer and you want to hit the ground running on what's new in ATL 8, Appendix C, "Moving to ATL 8," is for you. Finally, if you'd like an introduction to attributes (and a bit more information about why we've relegated attribute coverage to a lowly appendix), you'll want to read Appendix D, "Attributed ATL."

Conventions

When writing these chapters, it became necessary to show you not only diagrams and sample code, but also internal ATL implementation code. This book often becomes your personal tour guide through the ATL source code. To help you distinguish author-generated code from Microsoft-employee-generated code, we've adopted the following convention:

```
// This code is author-generated and is an example of what you'd type.
// Bold-faced code requires your particular attention.
CComBSTR bstr = OLESTR("Hello, World.");
```

```
// Code with a gray background is part of ATL or Windows.
CComBSTR(LPCOLESTR pSrc) { m_str = ::SysAllocString(pSrc); }
```

Because the ATL team didn't write its code to be published in book form, we often had to reformat it or even abbreviate it. Every effort has been made to retain the essence of the original code, but, as always, the ATL source code is the final arbiter.

Sample Code and Further Information

More information about this book, including the sample source code, is available at www.sellsbrothers.com/writing/atlbook. On that site, you'll also find contact information so that you can report errors or give feedback.

Acknowledgments

We have a large number of people to thank for their contributions to this book. Chris Sells would like to thank his wife, Melissa, and his boys, John and Tom, for sparing him countless evenings and weekends to work on this project. Chris would also like to thank Brent Rector for letting him horn in on the first edition of this book, Kirk Fertitta for updating a large portion of this book to ATL 7, and Chris Tavares for bringing this project home.

Brent would like to thank his wife, Lisa, and his children, Carly and Sean, for delaying the delivery of this book significantly. If it weren't for them, he would never have left the computer some days. Brent would also like to thank Chris Sells for his intelligence, patience, and general good looks.[2]

Kirk Fertitta would like to thank the following: the readers, who, after all, make book writing a worthwhile and rewarding endeavor; Chris Sells, for getting him involved in this book project and for his insights into ATL and into the writing process itself; Brad Handa and Hugues Valois, for countless hours working on real projects unraveling COM and ATL mysteries; all the subscribers to DevelopMentor's ATL discussion list for their sharing their perspectives and experiences with many of the new ATL features; MusicMatch (now Yahoo!) employees and contractors for all their feedback on using ATL in a very large commercial application (many of the caveats of using some of the attributed ATL features were exposed by this talented and patient group of developers); and Stephane Thomas, of Addison-Wesley, for her patience in getting this project started.

Chris Tavares would like to thank his long-suffering wife, Wendy, for her understanding, love, and support. The late-night glasses of water and bowls of ice cream were instrumental in finishing this book and keeping Chris sane. Thanks also go to his son, Matthew, who didn't mind too much when Daddy disappeared into his office for days at a time. Chris would also like to thank Chris Sells for the opportunity to help get the second edition out to the ATL community.

Chris, Kirk, Brent, and Chris would like to thank several folks together, starting first with the reviewers: Bill Craun, Johan Ericsson, Igor Tandetnik, Kim Gräsman,

[2] You get only one guess as to who wrote that part, and he doesn't have my initials or my good looks. BER

Jeff Galinovsky, Igor Tandetnik, and Nenad Stefanovic. Special thanks go to the members of the ATL team, including Christian Beaumont, Jim Springfield, Walter Sullivan, and Mark Kramer, for suffering nagging questions and taking the time to answer them. More special thanks to Don Box for his MSJ ATL feature, which so heavily influenced the ATL short-course and, in turn, this book. Thanks to reviewers Don Box, Keith Brown, Jon Flanders, Mike Francis, Kevin Jones, Stanley Lippman, Dharma Shukla, Jim Springfield, Jeff Stalls, Jaganathan Thangavelu, and Jason Whittington. Special thanks go to Dharma for his especially thorough and educational reviews. Thanks to Fritz Onion for his groundbreaking work delving into the depths of ATL control containment. Thanks to a former student, Valdan Vidakovic, for inspiring Chris to delve a bit more into the HTML control. Thanks to Tim Ewald, Jim Springfield, and Don Box for their help in developing the forwarding shims trick. Thanks to the members of the ATL and DCOM mailing lists, especially Don Box, Tim Ewald, Charlie Kindel, Valery Pryamikov, Mark Ryland, and Zane Thomas. Also, we'd like to thank George Shepherd for his initial research and even a little writing for the ATL Server chapters. And last, but not least, thanks to Addison-Wesley, especially Karen Gettman, Lori Lyons, and Kim Boedigheimer, for providing an environment in which we actually want to write (although not as quickly or as concisely as they might prefer. . .).

About the Authors

Chris Tavares is currently a software development engineer in the Microsoft patterns and practices group, where he strives to help developers learn the best way to develop on the Microsoft platform. He first touched a computer in third grade, doing hand-assembly of machine code on an Intel 8080 machine with 512 bytes (yes, bytes) of memory, a hex keypad, and 7 segment LCD display. He's been digging into computers and software ever since.

Kirk Fertitta is CTO of Pacific MindWorks, a leading provider of tools and services for electronic test and measurement. With his team at Pacific MindWorks, Kirk works extensively on code generation technology and Visual Studio extensibility. He is also a .NET/C# instructor for Pluralsight.

Brent Rector, president and founder of Wise Owl Consulting, is a noted speaker, consultant, and author, specializing in .NET, ASP.NET, XML, COM+, and ATL.

Chris Sells is a program manager for the Connected Systems Division. He's written several books, including *Programming Windows Presentation Foundation*, *Windows Forms Programming in C#*, and *ATL Internals*. In his free time, Chris hosts various conferences and makes a pest of himself on Microsoft internal product team discussion lists. More information about Chris, and his various projects, is available at http://www.sellsbrothers.com

This page intentionally left blank

CHAPTER 1

Hello, ATL

Welcome to the Active Template Library (hereafter referred to as ATL). In this chapter, I present a few of the tasks that you'll probably want to perform using ATL and the integrated wizards. This is by no means all of what ATL can accomplish, nor is it meant to be exhaustive coverage of the wizards or their output. In fact, the rest of this book focuses on how ATL is implemented to provide the Component Object Model (COM) glue that holds together this example (as well as several others). This chapter is actually just a warm-up to get you familiar with the support that the Visual Studio environment provides the ATL programmer.

What Is ATL?

Expanding the acronym doesn't completely describe what ATL is or why we have it. The *Active* part is actually residue from the marketing age at Microsoft, when "ActiveX"[1] meant all of COM. As of this writing, "ActiveX" means controls. And although ATL does provide extensive support for building controls, it offers much more than that.

ATL provides:

- Class wrappers around high-maintenance data types such as interface pointers, `VARIANT`s, `BSTR`s, and `HWND`s
- Classes that provide implementations of basic COM interfaces such as `IUnknown`, `IClassFactory`, `IDispatch`, `IPersistXxx`, `IConnectionPointContainer`, and `IEnumXxx`
- Classes for managing COM servers—that is, for exposing class objects, performing self-registration, and managing the server lifetime
- Classes for building COM controls and COM control containers, as well as for building plain old Windows applications

[1] The original expansion of ATL was ActiveX Template Library.

- An enormous library of classes for building web applications and XML web services
- Wizards, to save you typing

ATL was inspired by the current model citizen in the world of C++ class libraries, the C++ Standard Library. ATL is meant to be a set of small, efficient, and flexible classes. However, all this power requires a bit of skill to fully harness. As with the standard C++ library, only an experienced C++ programmer can use ATL effectively.

Of course, because we'll be programming COM, experience using and implementing COM objects and servers is absolutely required. For those of you hoping for a way to build your COM objects without COM knowledge, ATL is not for you (nor is Visual Basic, MFC, or anything else, for that matter). In fact, using ATL means being intimately familiar with COM in C++, as well as with some of the implementation details of ATL itself.

Still, ATL is packaged with several wizards that are helpful for generating the initial code. In the rest of this chapter, I present the various wizards available for ATL programmers as of Visual Studio 2005. Feel free to follow along.

Creating a COM Server

Creating an ATL Project

The first step in any Visual Studio development endeavor is to build the solution and the initial project. Choosing File, New Project displays the New Project dialog box shown in Figure 1.1, which focuses on Visual C++ projects.

Selecting the Visual C++ Projects folder displays the various types of C++ project templates available. The name of the project (shown in the figure as `PiSvr`) is the name of your generated DLL or EXE server.

The job of the ATL Project Template is to build a project for your COM server. A COM server is either a dynamic link library (DLL) or an executable (EXE). Furthermore, the EXE can be a standalone application or an NT service. The ATL Project Template supports all three of these server types. By default, a DLL server is selected as shown in Figure 1.2.

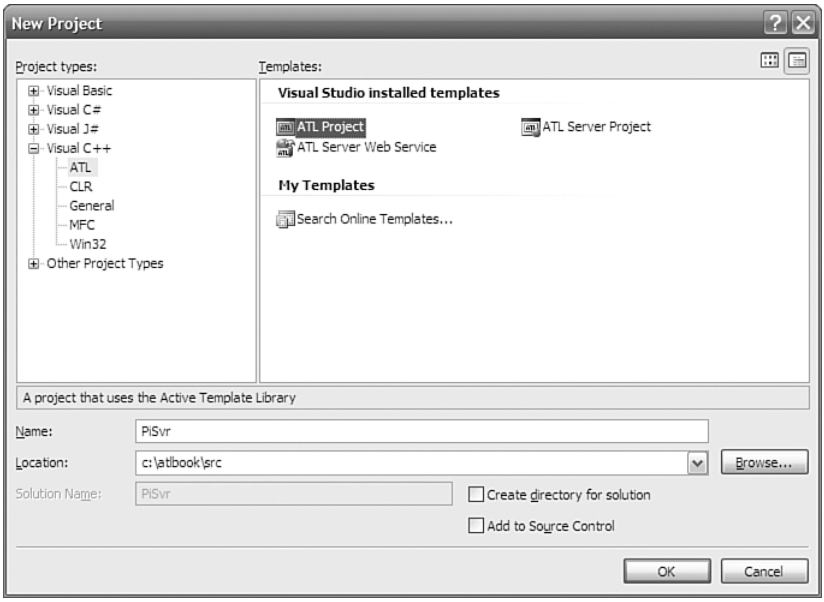

Figure 1.1 Creating a new Visual Studio project

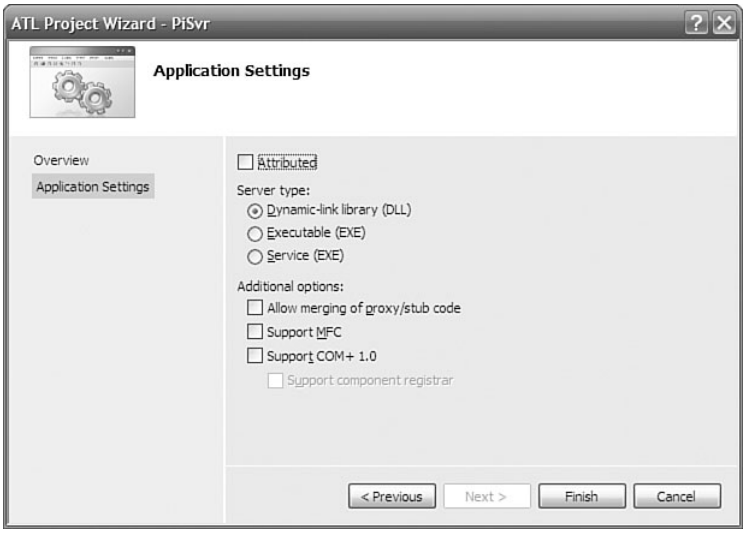

Figure 1.2 Project settings for an ATL project

ATL Project Wizard Options

The ATL Project Wizard in Figure 1.2 lists several options that merit discussion. The first option sets up the project to use ATL attributes. As described in the Preface, nonattributed projects are preferred in ATL 8, so we concentrate on nonattributed projects in this book. If you decide to use attributes anyway, see Appendix D, "Attributed ATL," for coverage of attributed projects.

In the Additional Options section, the first option enables you to bundle your custom proxy/stub code with your DLL server. By default, this option is not selected. As a result, Visual Studio generates a separate project named `<project-name>PS.vcproj` for the proxy/stub and adds it to the solution for your server. This project is for building a separate proxy/stub DLL to distribute to all client and server machines that need to marshal and unmarshal your custom interfaces. However, the proxy/stub project is not selected to be built in either of the default build configurations (Debug, Release) for the solution, as you can see from the property pages for the solution in Figure 1.3. Checking the Build check box next to the proxy/stub project causes this project to be built along with the main server whenever the solution is built.

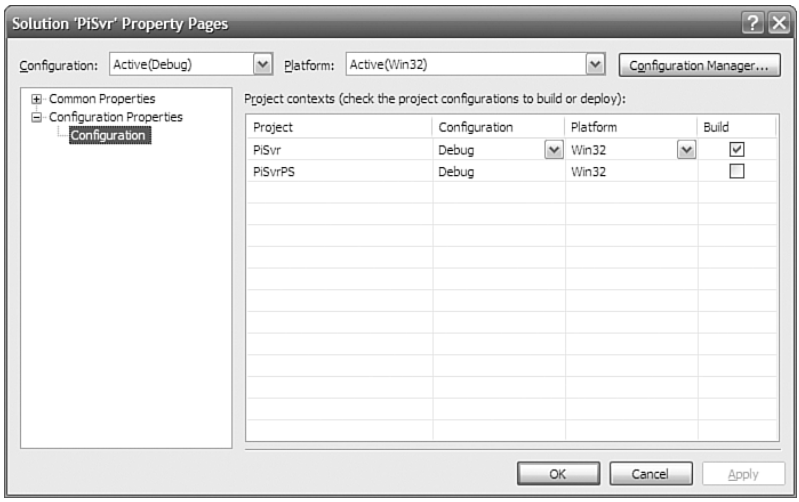

Figure 1.3 Application settings for a new ATL COM server

If you want to bundle the proxy/stub DLL into the server DLL (requiring the server to be installed on the client machine), you can check the Allow Merging of Proxy/Stub Code option (for nonattributed projects). Doing so results in a solution with only a single project (the one for your main server), with a bunch of conditionally compiled statements inserted into your server code to merge the proxy/stub code. The preprocessor definition `_MERGE_PROXYSTUB` controls whether the proxy/stub code is compiled into the server; this definition is added to all project configurations by default.

Unless you have a good reason (which is beyond the scope of this book), you'll want to avoid the option to merge the proxy/stub code into your server, instead preferring dual or oleautomation-compatible custom interfaces.

The second ATL Project Wizard option enables you to use the Microsoft Foundation Classes (MFC). Frankly, you should avoid this option. The following are a few common objections developers have to turning off this check box:

- **"I can't live without `CString` (or `CMap`, `CList`, and so on)."** The MFC utility classes were built as a stopgap until the C++ standards committee defined a standard library. They've done it, so we can stop using the MFC versions. The classes `string`, `map`, `list`, and so on provided in the standard library are more flexible and more robust than their MFC equivalents. Moreover, `CString` is now a shared class between MFC and ATL, and thus is available in ATL projects without having to include the rest of MFC and link to the MFC library. Other MFC utility classes have also been made shared, including CPoint and CRect. And for those who liked the MFC collections classes, ATL now includes MFC style collections (CAtlMap, CAtlList, and so on).
- **"I can't live without the wizards."** This chapter is all about the wizards that Visual Studio provides for ATL programmers. The ATL wizards are arguably as extensive as the ones MFC provides.
- **"I already know MFC, and I can't learn anything new."** Luckily, none of these people are reading this book.

The third ATL COM AppWizard option, Support COM+, causes your project to link to the COM+ component services library `comsvcs.dll` and includes the appropriate header file `comsvcs.h` so that your server can access the various interfaces that comprise COM+. With Support COM+ selected, you can also check the option Support Component Registrar, which generates an additional coclass in your project that implements the `IComponentRegistrar` interface.[2]

Results of the ATL Project Wizard

With or without these three options, every COM server that the ATL Project Wizard generates supports the three jobs of every COM server: self-registration, server lifetime control, and class objects exposure. As an additional convenience, the wizard adds a post-build event that registers the COM server upon each successful build. This step runs either `regsvr32.exe <project>.dll` or `<project>.exe /regserver`, depending on whether it is a DLL or EXE server.

[2] The `IComponentRegistrar` interface was thrown in for early implementations of COM+ but isn't actually used, as far as we know.

For more information about ATL's support for the three jobs of every COM server, as well as how you can extend it for more advanced concurrency and lifetime needs, see Chapter 5, "COM Servers."

Inserting a COM Class

Adding an ATL Simple Object

When you have an ATL COM server, you'll probably want to insert a new COM class. This is accomplished by selecting the Add Class item in the Project menu. When inserting an ATL class, you first have to choose the type of class you want, as shown in Figure 1.4.

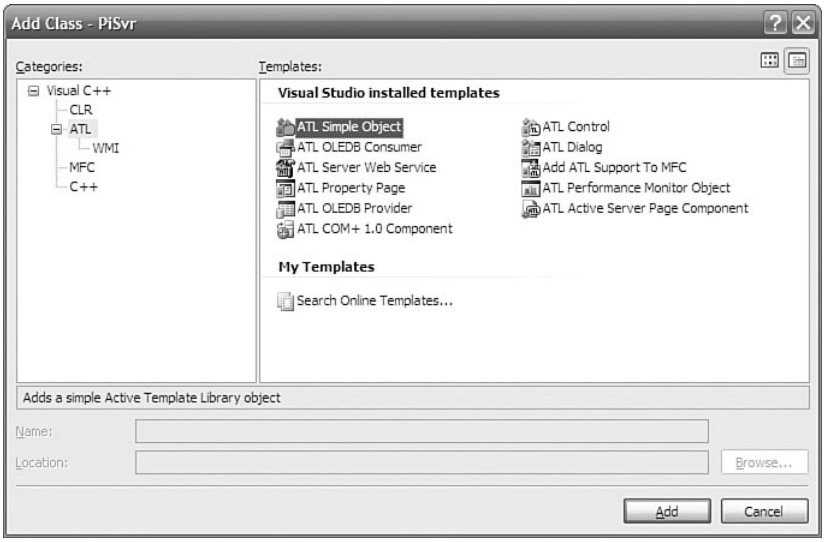

Figure 1.4 Adding an ATL COM class

If you're following along, you might want to take a moment to explore the various types of classes available. Each of these classes results in a specific set of code being generated, using the ATL base classes to provide most of the functionality and then generating the skeleton for your own custom functionality. The wizard for a particular class type is your chance to decide which interfaces you want your COM class to implement. Unfortunately, the wizard doesn't provide access to all the functionality of ATL (or even most of it), but the generated code is designed to be easy for you to add or subtract functionality after it has gotten you started. Experimentation is the best way to get familiar with the various wizard-generated class types and their options.

After you choose one of the class types (and press OK), Visual Studio generally asks for some specific information from you. Some of the classes have more options than the ATL Simple Object (as selected in Figure 1.4), but most of the COM classes require at least the information shown in Figures 1.5 and 1.6.[3]

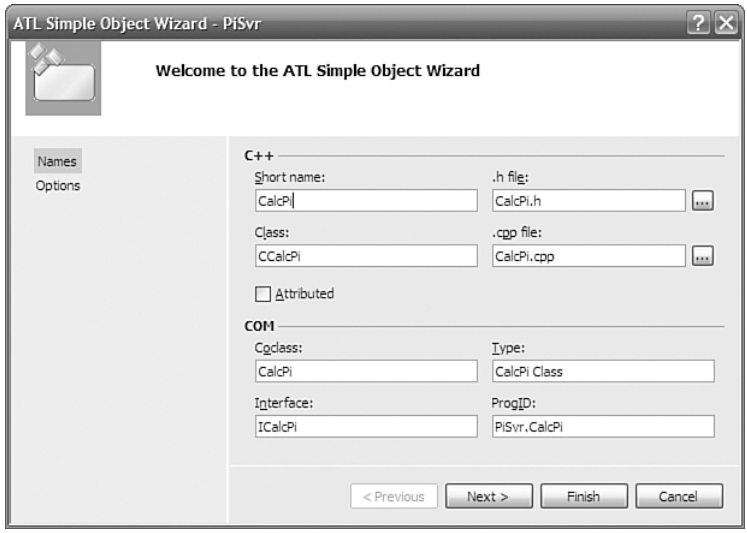

Figure 1.5 Setting COM class names

The Names tab of the ATL Simple Object Wizard dialog box requires you to type in only the short name, such as `CalcPi`. This short name is used to compose the rest of the information in this dialog box (which you can override, if you choose). The information is divided into two categories. The necessary C++ information is the name of the C++ class and the names of the header and implementation files. The necessary COM information is the `coclass` name (for the Interface Definition Language [IDL]); the name of the default interface (also for the IDL); the friendly name, called the Type (for the IDL and the registration settings); and finally the version-independent programmatic identifier (for the registration settings). The versioned ProgID is just the version-independent ProgID with the ".1" suffix. Note that the Attributed option is not selected. In a nonattributed project, we have the option of selectively adding attributed classes to our project via the Attributed check box in Figure 1.5.

[3] I should note that the ATL team got the terminology wrong here. The ATL Simple Object Wizard inserts a "class," not an "object."

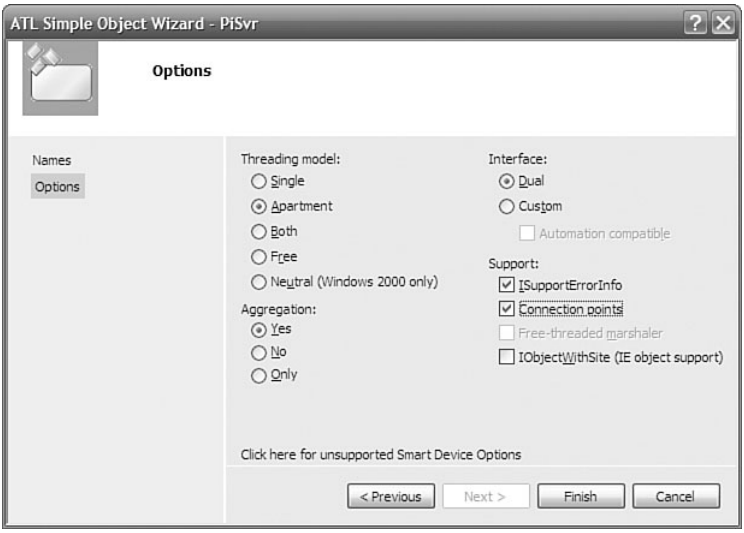

Figure 1.6 Setting COM class attributes

The Options page is your chance to make some lower-level COM decisions. The Threading model setting describes the kind of apartment where you want instances of this new class to live: a single-threaded apartment (STA, also known as the Apartment model), or a multithreaded apartment (MTA, also known as the Free-Threaded model). The Single model is for the rare class that requires all its objects to share the application's main STA, regardless of the client's apartment type. The Both model is for objects that you want to live in the same apartment as their clients, to avoid the overhead of a proxy/stub pair. The Neutral threading model is available only on Windows 2000 (and later) and is useful for objects that can safely execute on the thread of the caller. Calls to components marked as Neutral often execute more quickly because a thread switch is not required, whereas a thread switch always occurs with cross-apartment calls between other types of apartments. The Threading model setting that you choose determines the value for the `ThreadingModel` named value placed in the Registry for your server; it determines just how thread safe you need your object's implementation of `AddRef` and `Release` to be.

The Interface setting enables you to determine the kind of interface you want the class's default interface to be: Custom (it needs a custom proxy/stub and does not derive from `IDispatch`) or Dual (it uses the type library marshaler and derives from `IDispatch`). This setting determines how the IDL that defines your default interface is generated. Custom interfaces can further be qualified by selecting the Automation Compatible option. This option adorns the IDL interface definition with the `[oleautomation]` attribute, which restricts the variable types that your

interface's methods can use to OLE Automation-compatible types. For instance, `[oleautomation]` interface methods must use the `SAFEARRAY` type instead of conventional C-style arrays. If you plan to use your ATL COM object from several different client environments, such as Visual Basic, it is a good idea to check this option. Moreover, accessing COM objects from code running in the Microsoft .NET framework is much simpler if `[oleautomation]` interfaces are used. Deploying your COM object might also be simpler with the use of `[oleautomation]` interfaces because these interfaces always use the universal marshaler to pass interface references across apartments. The type library marshaler always is present on a machine that supports COM, so you don't need to distribute a proxy/stub DLL with your component.

The Aggregation setting enables you to determine whether you want your objects to be aggregatable—that is, whether to participate in aggregation as the controlled inner. This setting does not affect whether objects of your new class can use aggregation as the controlling outer. See Chapter 4, "Objects in ATL," for more details about being aggregated, and Chapter 6, "Interface Maps," about aggregating other objects.

The Support ISupportErrorInfo setting directs the wizard to generate an implementation of `ISupportErrorInfo`. This is necessary if you want to throw COM exceptions. COM exceptions (also called COM Error Information objects) enable you to pass more detailed error information across languages and apartment boundaries than can be provided with an `HRESULT` alone. See Chapter 5, "COM Servers," for more information about raising and catching COM exceptions.

The Support Connection Points setting directs the wizard to generate an implementation of `IConnectionPoint`, which allows your object to fire events into scripting environments such as those hosted by Internet Explorer. Controls also use connection points to fire events into control containers, as discussed in Chapter 9, "Connection Points."

The Help information for the Free-Threaded Marshaler setting reads as follows: "Allows clients in the same interface to get a raw interface even if their threading model doesn't match." This description doesn't begin to describe how dangerous it is for you to choose it. Unfortunately, the Free Threaded Marshaler (FTM) is like an expensive car: If you have to ask, you can't afford it. See Chapter 6, "Interface Maps," for a description of the FTM *before* checking this box.

The Support IObjectWithSite setting directs the wizard to generate an implementation of `IObjectWithSite`. Objects being hosted inside containers such as Internet Explorer use this interface. Containers use this interface to pass interface pointers to the objects they host so that these objects can directly communicate with their container.

Results of the ATL Simple Object Wizard

After you specify the options, the Simple Object Wizard generates the skeleton files for you to start adding your implementation. For the class, there is a newly generated header file containing the class definition, a .cpp file for the implementation, and an .RGS file containing registration information.[4] In addition, the IDL file is updated to contain the new interface definition.

The generated class definition looks like this:

```
// CalcPi.h : Declaration of the CCalcPi

#pragma once
#include "resource.h"       // main symbols

#include "PiSvr.h"
#include "_ICalcPiEvents_CP.h"

// CCalcPi

class ATL_NO_VTABLE CCalcPi :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CCalcPi, &CLSID_CalcPi>,
    public ISupportErrorInfo,
    public IConnectionPointContainerImpl<CCalcPi>,
    public CProxy_ICalcPiEvents<CCalcPi>,
    public IDispatchImpl<ICalcPi, &IID_ICalcPi, &LIBID_PiSvrLib,
        /*wMajor =*/ 1, /*wMinor =*/ 0>
{
public:
    CCalcPi() { }
DECLARE_REGISTRY_RESOURCEID(IDR_CALCPI)

BEGIN_COM_MAP(CCalcPi)
    COM_INTERFACE_ENTRY(ICalcPi)
    COM_INTERFACE_ENTRY(IDispatch)
    COM_INTERFACE_ENTRY(ISupportErrorInfo)
    COM_INTERFACE_ENTRY(IConnectionPointContainer)
END_COM_MAP()

BEGIN_CONNECTION_POINT_MAP(CCalcPi)
    CONNECTION_POINT_ENTRY(__uuidof(_ICalcPiEvents))
```

[4] See Chapters 4 and 5 for more information on COM class registration.

```
END_CONNECTION_POINT_MAP()
// ISupportsErrorInfo
    STDMETHOD(InterfaceSupportsErrorInfo)(REFIID riid);

    DECLARE_PROTECT_FINAL_CONSTRUCT()

    HRESULT FinalConstruct() {
        return S_OK;
    }

    void FinalRelease() {
    }

public:

};

OBJECT_ENTRY_AUTO(__uuidof(CalcPi), CCalcPi)
```

The first thing to notice is the list of base classes. In this instance, ATL takes advantage of both templates and multiple inheritance. Each base class provides a separate piece of the common code needed for a COM object:

- `CComObjectRootEx` provides the implementation of the `IUnknown` interface.
- `CComCoClass` provides the class factory implementation.
- `ISupportErrorInfo` is the interface; implementation for the one method is in the .cpp file.
- `IConnectionPointContainerImpl` provides the implementation I requested by checking the Support Connection Points check box.
- `CProxy_ICalcPiEvents` is part of the connection point implementation.
- `IDispatchImpl` provides the implementation of `IDispatch` needed for the object's dual interface.

The other important thing to note here is the `COM_MAP` macros. This is an instance of an ATL *map*: a set of macros that generate code (typically to fill in a lookup table). The `COM_MAP`, in particular, is used to implement the `QueryInterface` method that all COM objects are required to support.

For more information about the base classes that ATL uses to implement basic COM functionality and how you can leverage this implementation for building object hierarchies and properly synchronizing multithreaded objects, see Chapter 4, "Objects in ATL." For more information about how to make full use of the `COM_MAP`, see Chapter 6, "Interface Maps."

Adding Properties and Methods

One of the things that make a C++ programmer's life hard is the separation of the class declaration (usually in the `.h` file) and the class definition (usually in the `.cpp` file). This can be a pain because of the maintenance required between the two. Any time a member function is added in one, it has to be replicated in the other. Manually, this can be a tedious process, and it is made even more tedious for a C++ COM programmer who must maintain the same definitions in an `.idl` file. When I'm adding properties and methods to my interfaces, I'd like my C++ development environment to help translate an IDL method definition into C++ (with the appropriate ATL attributes, if necessary) and drop it into my `.h` and `.cpp` files for me, leaving me a nice place to provide my implementation. That's just what Visual Studio provides.

By right-clicking on a COM interface in Class view, you can choose to add a new property or method from the Add submenu of the context menu that appears. Figure 1.7 shows the dialog box that enables you to add a property to a COM interface. Parameters to the property can be added by specifying the parameter data type and the parameter direction (for example, `[in]` or `[out]`).

Figure 1.7 Adding a property

Figure 1.8 shows the options available on the IDL Attributes tab for the Add Property Wizard. Selected attributes are inserted into the appropriate interface definition, in your project's IDL file. In either case, the effect on the type library is identical. Many of these attributes apply in rare circumstances, so the default selections and values shown in the figure are often suitable. In any event, adding, deleting, or modifying these attributes directly in the IDL file afterward is a simple matter.

Figure 1.8 IDL attributes for a property

The following shaded code shows the implementation skeleton that the wizard generates. We have to provide only the appropriate behavior (shown as unshaded code).

```
STDMETHODIMP CCalcPi::get_Digits(LONG* pVal) {
  *pVal = m_nDigits;
  return S_OK;
}

STDMETHODIMP CCalcPi::put_Digits(LONG newVal) {
  if( newVal < 0 )
    return Error(L"Can't calculate negative digits of PI");
  m_nDigits = newVal;
    return S_OK;
}
```

Similarly, we can add a method by right-clicking an interface in Class view and choosing Add Method. Figure 1.9 shows the Add Method Wizard. Input and output parameters are added individually using the Parameter Type combo box, the Parameter Name text box, and the Add/Remove buttons.

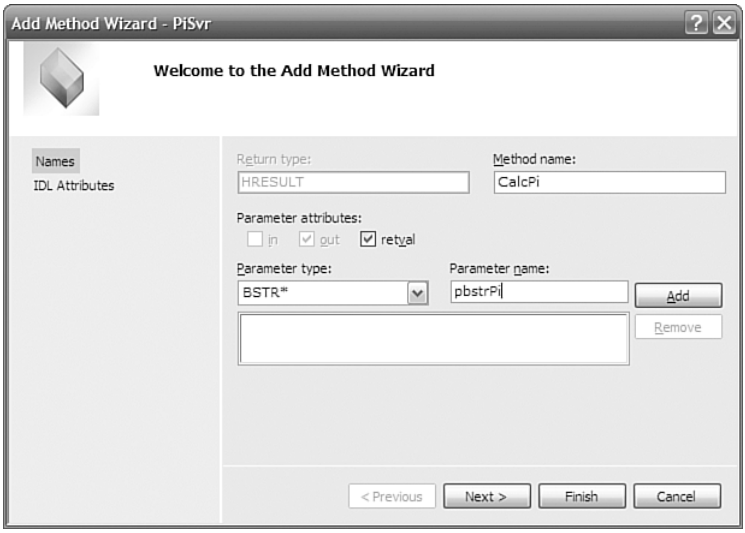

Figure 1.9 Adding a method

Again, the wizard updates the interface definition in either the IDL file or the header file, generates the appropriate C++ code, and places us in the implementation skeleton to do our job. The shaded code is what remains of the wizard-generated C++ code after I added the code to implement the method:

```
STDMETHODIMP CCalcPi::CalcPi(BSTR* pbstrPi) {
  _ASSERTE(m_nDigits >= 0);

  if( m_nDigits ) {
    *pbstrPi = SysAllocStringLen(L"3.", m_nDigits+2);
    if( *pbstrPi ) {
      for( int i = 0; i < m_nDigits; i += 9 ) {
        long nNineDigits = NineDigitsOfPiStartingAt(i+1);
        swprintf(*pbstrPi + i+2, 10, L"%09d", nNineDigits);
      }

      // Truncate to number of digits
      (*pbstrPi)[m_nDigits+2] = 0;
    }
  }
  else {
    *pbstrPi = SysAllocString(L"3");
  }

  return *pbstrPi ? S_OK : E_OUTOFMEMORY;
}
```

For a description of COM exceptions and the ATL `Error` function (used in the `put_Digits` member function), see Chapter 4, "Objects in ATL."

Implementing Additional Interfaces

Interfaces are the core of COM, and most COM objects implement more than one. Even the wizard-generated ATL Simple Object, shown earlier, implements four interfaces (one custom interface and three standard interfaces). If you want your ATL-based COM class to implement another interface, you must first have a definition of it. For example, you can add the following interface definition to your project's IDL file:

```
[
    object,
    uuid("27ABEF5D-654F-4D85-81C7-CC3F06AC5693"),
    helpstring("IAdvertiseMyself Interface"),
    pointer_default(unique)
]
interface IAdvertiseMyself : IUnknown {
    [helpstring("method ShowAd")]
    HRESULT ShowAd(BSTR bstrClient);
};
```

To implement this interface in your project, you simply add the new interface to your C++ class inheritance list and add the interface to the `COM_MAP`:

```
class ATL_NO_VTABLE CCalcPi :
    public ICalcPi,
    public IAdvertiseMyself {

BEGIN_COM_MAP(CCalcPi)
    COM_INTERFACE_ENTRY(ICalcPi)
    COM_INTERFACE_ENTRY(IAdvertiseMyself)
    ...
END_COM_MAP()
```

If methods in the `IAdvertiseMyself` interface need to throw COM exceptions, the generated implementation of `ISupportErrorInfo` must be modified as well. This is accomplished by simply adding the IID to the array in the generated implementation:

```
STDMETHODIMP CCalcPi::InterfaceSupportsErrorInfo(REFIID riid) {
    static const IID* arr[] = {
        &IID_ICalcPi,
        &IID_IAdvertiseMyself
    };

    for (int i=0; i < sizeof(arr) / sizeof(arr[0]); i++) {
        if (InlineIsEqualGUID(*arr[i],riid))
            return S_OK;
    }
    return S_FALSE;
}
```

After you make the updates, you have to implement the new interface's methods.

```
STDMETHODIMP CCalcPi::ShowAd(BSTR bstrClient) {
  CComBSTR  bstrCaption = OLESTR("CalcPi hosted by ");
  bstrCaption += (bstrClient && *bstrClient ?
    bstrClient : OLESTR("no one"));
  CComBSTR  bstrText =
    OLESTR("These digits of pi brought to you by CalcPi!");

  MessageBox(0, COLE2CT(bstrText), COLE2CT(bstrCaption),
    MB_SETFOREGROUND);

  return S_OK;
}
```

VS provides a convenient wizard to make this process simpler. Right-clicking the class from Class View and selecting Implement Interface from the Add submenu brings up the Implement Interface Wizard, shown in Figure 1.10. This wizard enables you to implement interfaces defined in an existing type library. The wizard is smart enough to pull type libraries from the current project. Alternatively, you can define interfaces in IDL, compile them using MIDL, and implement those interfaces by referencing the resulting type library. The radio buttons enable you to use the type library from the current project, registered type libraries (Registry), or unregistered type libraries (File) that you locate with the Browse button. For our PiSvr project, the type library built from the generated IDL makes the interfaces we've defined available to the Implement Interface Wizard.

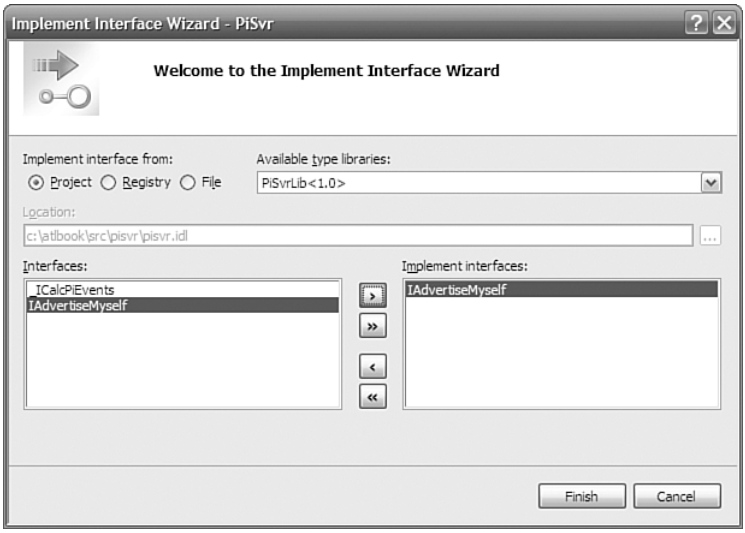

Figure 1.10 The Implement Interface Wizard

Note that interfaces that have already been implemented (`ICalcPi`, in our case) do not appear in the list of implementable interfaces. Unfortunately, the Implement Interface Wizard does not support interfaces that don't exist in type libraries; this leaves out most of the standard COM interfaces, such as `IPersist`, `IMarshal`, and `IOleItemContainer`.

Unfortunately, there's a bug in this wizard. The wizard did this to our base class list:

```
class ATL_NO_VTABLE CCalcPi :
    ... the usual stuff …
    public IDispatchImpl<ICalcPi, &IID_ICalcPi, &LIBID_PiSvrLib,
        /*wMajor =*/ 1, /*wMinor =*/ 0>,
    public IDispatchImpl<IAdvertiseMyself,
        &__uuidof(IAdvertiseMyself), &LIBID_PiSvrLib,
        /* wMajor = */ 1, /* wMinor = */ 0>
{
...
```

The added code is in bold. The wizard added the `IDispatchImpl` template as a base class. This is used when implementing dual interfaces. `IAdvertiseMyself` is *not* a dual interface. The wizard should have just derived from the interface directly. The fix is easy: Change the previous bold line to this:

```
    public IAdvertiseMyself
```

Even with this bug, the Implement Interface Wizard is still worth using for large interfaces. In addition to updating the base class list and the COM_MAP, the wizard provides skeleton implementation for all the methods in the interface; for a large interface, this can save a ton of typing. Unfortunately, the skeletons are added only to the header file, not to the .cpp file.

For more information about the various ways that ATL allows your COM classes to implement interfaces, see Chapter 6, "Interface Maps." For more information about CCom-BSTR and the string-conversion routines used in the ShowAd method, see Chapter 2, "Strings and Text."

Support for Scripting

Any time you run the ATL Simple Object Wizard and choose Dual as the interface type, ATL generates an interface definition for the default interface that derives from IDispatch and is marked with the dual attribute, and places it in the IDL file. Because it derives from IDispatch, our dual interface can be used by scripting clients such as Active Server Pages (ASP), Internet Explorer (IE), and Windows Script Host (WSH). When our COM class supports IDispatch, we can use objects of that class from scripting environments. Here's an example HTML page that uses an instance of the CalcPi object:

```
<object classid="clsid:859512CF-E4D8-450C-AF09-6578FE2F6DC2"
        id=objPiCalculator>
</object>

<script language=vbscript>
  ' Set the digits property
  objPiCalculator.digits = 5

  ' Calculate pi
  dim pi
  pi = objPiCalculator.CalcPi

  ' Tell the world!
  document.write "Pi to " & objPiCalculator.digits & _
    " digits is " & pi
</script>
```

For more information about how to handle the inconvenient data types associated with scripting—namely, BSTRs and VARIANTs—see Chapter 2, "Text and Strings," and Chapter 3, "ATL Smart Types."

Adding Persistence

ATL provides base classes for objects that want to be persistent—that is, saved to some persistence medium (such as a disk) and restored later. COM objects expose this support by implementing one of the COM persistence interfaces, such as `IPersistStreamInit`, `IPersistStorage`, or `IPersistPropertyBag`. ATL provides implementation of these three persistence interfaces—namely, `IPersistStreamInitImpl`, `IPersistStorageImpl`, and `IPersistPropertyBagImpl`. Your COM object supports persistence by deriving from any of these base classes, adding the interface to your `COM_MAP`, and adding a data member called `m_bRequiresSave` that each of these base classes expects.

```
class ATL_NO_VTABLE CCalcPi :
  public ICalcPi,
  public IAdvertiseMyself,
  public IPersistPropertyBagImpl<CCalcPi> {
public:
  ...

  // ICalcPi
public:
  STDMETHOD(CalcPi)(/*[out, retval]*/ BSTR* pbstrPi);
  STDMETHOD(get_Digits)(/*[out, retval]*/ long *pVal);
  STDMETHOD(put_Digits)(/*[in]*/ long newVal);

public:
  BOOL m_bRequiresSave; // Used by persistence base classes

private:
  long m_nDigits;
};
```

However, that's not quite all there is to it. ATL's implementation of persistence needs to know which parts of your object need to be saved and restored. For that information, ATL's implementations of the persistent interfaces rely on a table of object properties that you want to persist between sessions. This table, called a `PROP_MAP`, contains a mapping of property names and dispatch identifiers (as defined in the IDL file). So, given the following interface:

```
[
object,
...
]
```

```
interface ICalcPi : IDispatch {
    [propget, id(1)] HRESULT Digits([out, retval] LONG* pVal);
    [propput, id(1)] HRESULT Digits([in] LONG newVal);
};
```

the PROP_MAP would be contained inside our implementation of ICalcPi like this:

```
class ATL_NO_VTABLE CCalcPi : ...
{
  ...
public:
BEGIN_PROP_MAP(CCalcPi)
  PROP_ENTRY("Digits", 1, CLSID_NULL)
END_PROP_MAP()
};
```

Given an implementation of IPersistPropertyBag, our IE sample code can be expanded to support initialization of object properties via persistence using the <param> tag:

```
<object classid="clsid:E5F91723-E7AD-4596-AC90-17586D400BF7"
        id=objPiCalculator>
        <param name=digits value=5>
</object>

<script language=vbscript>
  ' Calculate pi
  dim pi
  pi = objPiCalculator.CalcPi

  ' Tell the world!
  document.write "Pi to " & objPiCalculator.digits & _
    " digits is " & pi
</script>
```

For more information about ATL's implementation of persistence, see Chapter 7, "Persistence in ATL."

Adding and Firing Events

When something interesting happens in a COM object, we'd like to be able to spontaneously notify its client without the client polling the object. COM provides a standard mechanism for sending these notifications to clients (normally called "firing an event") using the connection-point architecture.

Connection-point events are actually methods on an interface. To support the widest variety of clients, an event interface is often defined as a `dispinterface`. Choosing Support Connection Points in the ATL Simple Object Wizard generates an event in our IDL file. The following is an example of the wizard-generated code augmented with a single event method (shown in bold):

```
[
    uuid(B830F523-D87B-434F-933A-623CEF6FC4AA),
    helpstring("_ICalcPiEvents Interface")
]
dispinterface _ICalcPiEvents {
    properties:
    methods:
    [id(1)] void OnDigit([in] short nIndex,
        [in] short nDigit);
};
```

In addition to changing the IDL file, the Support Connection Points option makes several changes to the class definition. The `IConnectionPointContainerImpl` base class is added. This class implements the `IConnectionPointContainer` interface, providing functionality for managing multiple event interfaces on the class. The `IConnectionPointImpl` base class implements a connection point for a specific event interface: `_ICalcPiEvents`, in this case. The `COM_MAP` is also modified to include an entry for `IConnectionPointContainer`, and a new map, the `CONNECTION_MAP`, is added to the class.

The wizard also generates a proxy class for the connection point. This proxy class is added to the base class list and provides a convenient way to actually fire the events (that is, call the methods on the connection point). This is very helpful because the typical connection point is a `dispinterface`.

For example:

```
STDMETHODIMP CCalcPi::CalcPi(BSTR *pbstrPi) {
  // (code to calculate pi removed for clarity)
  ...
```

```
  // Fire each digit
  for( short j = 0; j != m_nDigits; ++j ) {
    Fire_OnDigit(j, (*pbstrPi)[j+2] - L'0');
  }

  ...
}
```

Objects of the CCalcPi class can now send events that can be handled in a page of HTML:

```
<object classid="clsid:E5F91723-E7AD-4596-AC90-17586D400BF7"
        id=objPiCalculator>
        <param name=digits value=50>
</object>

<input type=button name=cmdCalcPi value="Pi to 50 Digits:">
<span id=spanPi>unknown</span>

<p>Distribution of first 50 digits in pi:
<table border cellpadding=4>
... <!- table code removed for clarity –>
</table>

<script language=vbscript>
  ' Handle button click event
  sub cmdCalcPi_onClick
    spanPi.innerText = objPiCalculator.CalcPi
  end sub

  ' Handle calculator digit event
  sub objPiCalculator_onDigit(index, digit)
    select case digit
    case 0: span0.innerText = span0.innerText + 1
    case 1: span1.innerText = span1.innerText + 1
    ... <!– etc –>
    end select
    spanTotal.innerText = spanTotal.innerText + 1
  end sub
</script>
```

The sample HTML page handles these events to provide the first 50 digits of pi and their distribution, as shown in Figure 1.11.

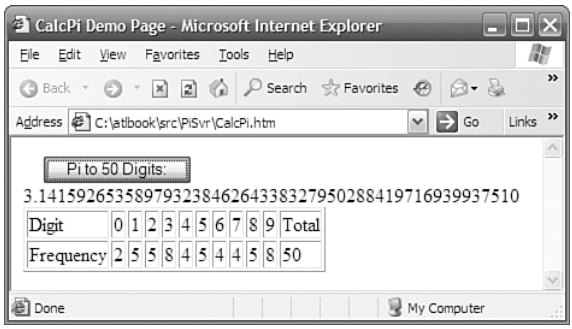

Figure 1.11 Pi to 50 digits

For more information about ATL's support for connection points, see Chapter 9, "Connection Points."

Using a Window

Because this is Microsoft Windows we're developing for, sometimes it's handy to be able to put up a window or a dialog box. For example, the `MessageBox` call we made earlier yielded a somewhat boring advertisement, as shown in Figure 1.12.

Figure 1.12 Boring message box

Normally, putting up a custom dialog box is kind of a pain. For the average Win32 programmer, either it involves lots of procedural code, which we don't like, or it involves building a bunch of forwarding code to map Windows messages to member functions (a dialog box is an object, after all). As with MFC, ATL has a great deal of functionality for building windows and dialog boxes. To add a new dialog box, select Add Class from the Project menu and then select ATL Dialog from the list of available templates, as shown in Figure 1.13.

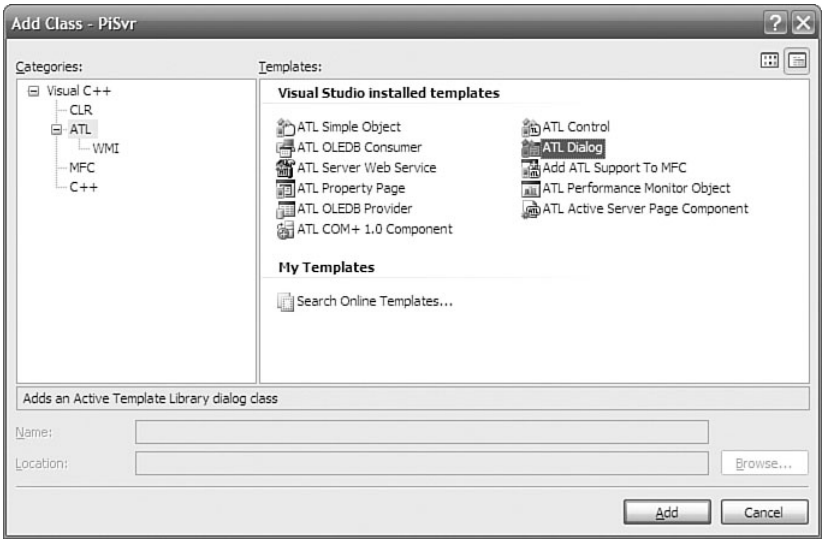

Figure 1.13 Inserting a dialog box class

The ATL Dialog Wizard (see Figure 1.14) is much simpler than many other ATL class templates. It allows you to enter only C++ name information because a dialog box is a Win32 object, not a COM object.

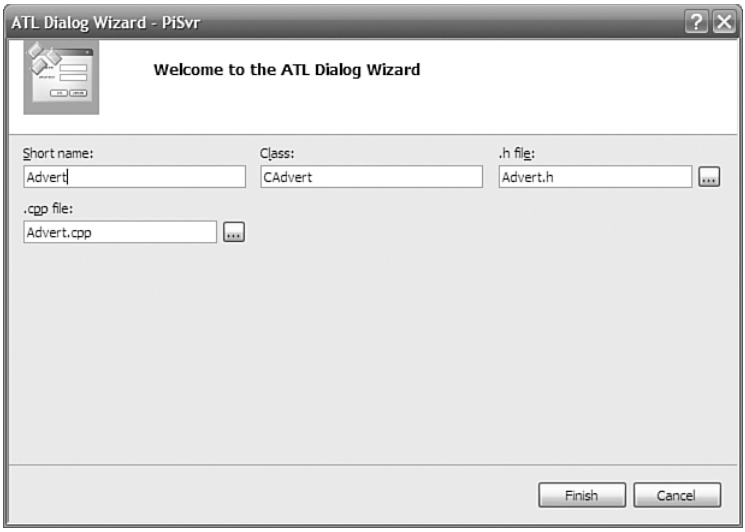

Figure 1.14 ATL Dialog Wizard

The generated code creates a class that derives from `CAxDialogImpl` and uses a new dialog box resource, also provided by the wizard. The derived class routes messages to handlers using the `MSG_MAP` macros, as shown here:

```
class CAdvert : public CAxDialogImpl<CAdvert> {
public:
  CAdvert() {}
  ~CAdvert() {}
  enum { IDD = IDD_ADVERT };

BEGIN_MSG_MAP(CAdvert)
    MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
    COMMAND_HANDLER(IDOK, BN_CLICKED, OnClickedOK)
    COMMAND_HANDLER(IDCANCEL, BN_CLICKED, OnClickedCancel)
    CHAIN_MSG_MAP(CAxDialogImpl<CAdvert>)
END_MSG_MAP()

  LRESULT OnInitDialog(UINT uMsg, WPARAM wParam, LPARAM lParam,
                       BOOL& bHandled) {
    if( m_bstrClient.Length() ) {
      CComBSTR  bstrCaption = OLESTR("CalcPi sponsored by ");
      bstrCaption += m_bstrClient;

      USES_CONVERSION;
      SetWindowText(OLE2CT(bstrCaption));
    }
    return 1;  // Let the system set the focus
  }

  LRESULT OnClickedOK(WORD wNotifyCode, WORD wID, HWND hWndCtl,
    BOOL& bHandled) {
    EndDialog(wID);
    return 0;
  }

  LRESULT OnClickedCancel(WORD wNotifyCode, WORD wID,
    HWND hWndCtl, BOOL& bHandled) {
    EndDialog(wID);
    return 0;
  }

  CComBSTR  m_bstrClient;
};
```

If you want to handle another message, you can add the appropriate entries to the message map and add the handler member functions by hand. If you prefer, you can add a message handler by right-clicking the name of the `CAxDialogImpl`-based class in Class view, choosing Properties, and clicking the Messages toolbar button. Figure 1.15 shows the resulting window.

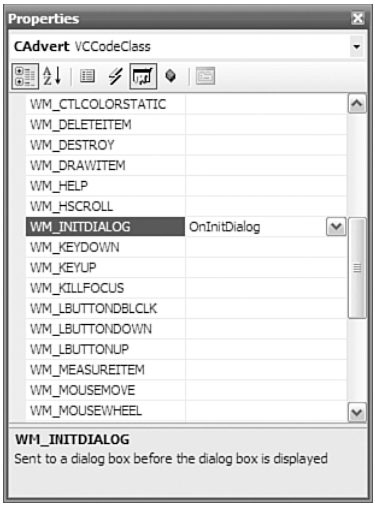

Figure 1.15 Adding a Windows message handler

For more information on ATL's extensive support for windowing, including building standalone Windows applications, see Chapter 10, "Windowing."

COM Controls

COM controls are objects that provide their own user interface (UI), which is closely integrated with that of their container. ATL provides extensive support for COM controls via the `CComControl` base class, as well as various other base `IXxxImpl` classes. These base classes handle most of the details of being a basic control (although there's plenty of room for advanced features, as shown in Chapter 11, "ActiveX Controls"). Had you chosen ATL Control from the Add Class dialog box when generating the `CalcPi` class, you could have provided the UI merely by implementing the `OnDraw` function:

```
HRESULT CCalcPi::OnDraw(ATL_DRAWINFO& di) {
  CComBSTR  bstrPi;
  if( SUCCEEDED(this->CalcPi(&bstrPi)) ) {
```

```
    DrawText(di.hdcDraw, COLE2CT(bstrPi), -1,
      (RECT*)di.prcBounds,
      DT_SINGLELINE | DT_CENTER | DT_VCENTER);
  }

  return S_OK;
}
```

The wizard would also have generated a sample HTML page, which I've augmented to take up the entire browser window and to set the initial number of digits to 50:

```
<HTML>
<HEAD>
<TITLE>ATL 8.0 test page for object CalcPiControl</TITLE>
</HEAD>
<BODY>

<OBJECT ID="CalcPi"
    CLASSID="CLSID:9E7ABA7A-C106-4813-A50C-B15C967264B6"
    height="100%" width="100%">
    <param name="Digits" value="50">
</OBJECT>

</BODY>
</HTML>
```

Displaying this sample page in Internet Explorer yields a view of a control (see Figure 1.16). For more information about building controls in ATL, see Chapter 11, "ActiveX Controls."

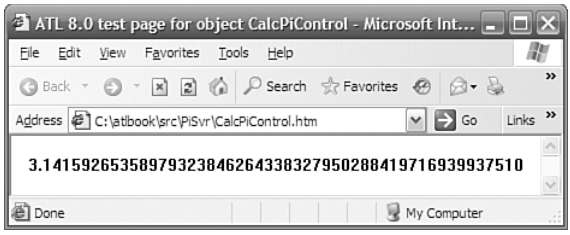

Figure 1.16 The `CalcPi` control hosted in Internet Explorer

Hosting a Control

If you want to host a control, you can do so with ATL's control-hosting support. For example, the `Ax` in `CAxDialogImpl` stands for ActiveX control and indicates that the dialog box is capable of hosting controls. To host a control in a dialog box, right-click the dialog box resource and choose Insert ActiveX Control.[5] This produces a dialog box that lists the controls installed on your system, as shown in Figure 1.17.

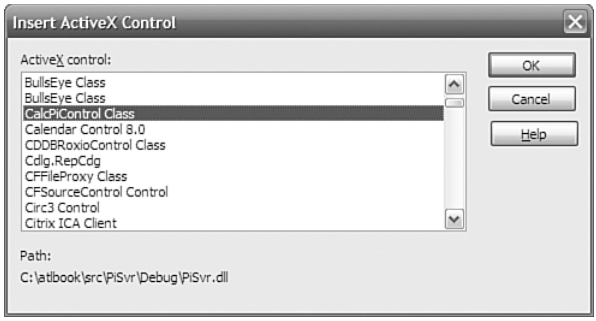

Figure 1.17 Insert ActiveX Control dialog box

After you insert the control, you can click on it and set its properties in the Properties window, as shown in Figure 1.18.

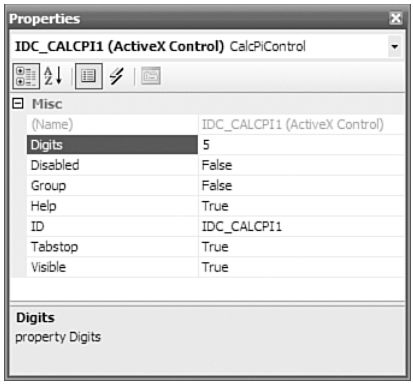

Figure 1.18 Control Properties dialog box

[5] For commonly used ActiveX controls, it's usually easier to add them to your Visual Studio toolbox than to go through this dialog box every time. Right-click the toolbox, select Choose Items, wait approximately 37 minutes for the dialog box to appear, select the COM Components tab, and select the controls you want on your toolbox.

By clicking the Control Events toolbar button, you also can choose to handle a control's events, as shown in Figure 1.19.

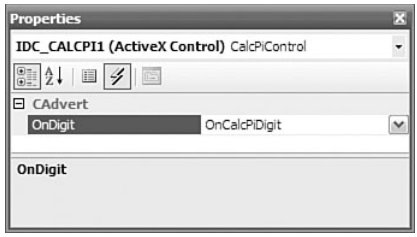

Figure 1.19 Choosing which control events to handle

When the dialog box is shown, the control is created and initialized based on the properties set at development time. Figure 1.20 shows an example of a dialog box hosting a control.

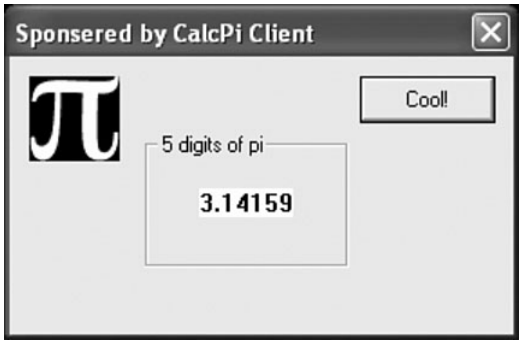

Figure 1.20 A dialog box hosting a COM control

ATL provides support for hosting ATL controls not only in dialog boxes, but also in other windows, in controls that have a UI declared as a dialog box resource (called composite controls), and in controls that have a UI declared as an HTML resource (called HTML controls). For more information about control containment, see Chapter 12, "Control Containment."

Being a C++ COM Client

COM and C++ go hand in hand—at least, theoretically. A COM interface maps directly to a C++ abstract class. All you need to do to use a COM object is run its IDL file through the MIDL compiler, and you've got a header file with all the information you need.

This worked well until the VB team asked if it could play with this COM stuff, too.

VB developers generally neither know nor want to know C++. And IDL is a language that's very much in the C++ tradition, with lots of support for C/C++-specific things in it (such as arrays and pointers) VB needed a way to store type information about COM objects that VB developers could use and understand easily.

Thus was born the *type library* (a.k.a. *typelib*). A typelib stores information about a COM object: The classid, the interfaces that the object supports, the methods on those interfaces, and so on—just about everything you'd find in an IDL file (with some unfortunate exceptions, mostly having to do with C-style arrays). The COM system includes a set of COM objects that lets you programmatically walk through the contents of a typelib. Best of all, the typelib can be embedded into a DLL or EXE file directly, so you never have to worry about the type information getting lost.

The typelib was so successful for VB developers that many COM components these days aren't shipped with an IDL file; the type library includes everything needed to use the components. Only one thing is missing: How do we use typelibs in C++?

The C++ language doesn't understand typelibs. It wants header files. This was such a serious problem that, back in Visual Studio 6, Microsoft extended the compiler so that it could use type libraries in much the same way that you use header files. This extension was the `#import` statement.

`#import` is used much like `#include` is. The general form is shown here:

```
#import "pisvr.dll" <options>
```

The `#import` statement generates either one or two C++ header files, depending on the options you use. These header files have the extensions `.tlh` (for "typelib header") and `.tli` (for "typelib inline") and are generated into your project output directory (by default, Debug for a debug build, Release for a release build).

The options on the `#import` line give you a great deal of control over the contents of the generated files. Check the Visual Studio documentation for the full list; we talk about some of the more commonly used options here.

The `no_namespace` option tells the compiler that we don't want the contents of the generated files to be placed into a C++ namespace. By default, the contents of the generated files are placed in a C++ namespace named after the type library.

`named_guids` instructs the compiler that we want to have named symbols for the GUIDs in the type library. By default, this would not compile because the name `CLSID_PISvr` would not be defined:

```
::CoCreateInstance( CLSID_PISvr, ... );
```

Instead, you have to do this:

```
::CoCreateInstance( __uuidof( PISvr ), ... );
```

You also need to use `__uuidof( )` to get the IID for interfaces.

The `raw_interfaces_only` option requires the most explanation. By default, when the `#import` statement generates the header file, it doesn't just spit out class definitions for interfaces. It actually generates wrapper classes that attempt to make a COM interface easier to use. For example, given the interface:

```
interface ICalcPi : IDispatch {
  [propget, id(1), helpstring("property Digits")]
  HRESULT Digits([out, retval] LONG* pVal);
  [propput, id(1), helpstring("property Digits")]
  HRESULT Digits([in] LONG newVal);
  [id(2), helpstring("method CalcPi")]
  HRESULT CalcPi([out,retval] BSTR* pbstrPi);
};
```

Normal use of this interface would be something like this:

```
HRESULT DoStuff( long nDigits, ICalcPi *pCalc ) {
    HRESULT hr = pCalc->put_Digits( nDigits );
    if( FAILED( hr ) ) return hr;

    BSTR bstrResult;
    hr = pCalc->CalcPi( &bstrResult );
    if( FAILED( hr ) ) return hr;

    std::cout << "PI to " << nDigits << " digits is "
        << CW2A( bstrResult );

    ::SysFreeString( bstrResult );
    return S_OK;
}
```

When using the `#import` statement, on the other hand, using this interface looks like this:

```
void DoStuff( long nDigits, ICalcPiPtr spCalc ) {
  spCalc->Digits = nDigits;
  _bstr_t bstrResults = spCalc->CalcPi();
  std::cout << "PI to " << spCalc->Digits << " digits is "
    << ( char * )bstrResults;
}
```

The `ICalcPiPtr` type is a smart pointer expressed as a typedef for the `_com_ptr_t` class. This class is *not* part of ATL; it's part of the Direct-To-COM extensions to the compiler and is defined in the system header file `comdef.h` (along with all the other types used by the wrapper classes). The smart pointer automatically manages the reference counting, and the `_bstr_t` type manages the memory for a `BSTR` (which we discuss in Chapter 2, "Strings and Text").

The most remarkable thing about the wrapper classes is that the `HRESULT` testing is gone. Instead, the wrapper class translates any failed `HRESULT`s into a C++ exception (the `_com_error` class, to be precise). This lets the generated code use the method's `[retval]` variable as the actual return value, which eliminates a lot of temporary variables and output parameters.

The wrapper classes can immensely simplify writing COM clients, but they have their downsides. The biggest is that they require the use of C++ exceptions. Some projects aren't willing to pay the performance penalties that exception handling brings, and throwing exceptions means that developers have to pay very careful attention to exception safety.

Another downside to the wrappers for ATL developers is that ATL also has wrapper classes for COM interfaces (see Chapter 3, "ATL Smart Types") and `BSTR`s (see Chapter 2). The ATL wrappers are arguably better than the ones defined in `comdef.h`; for example, you can accidentally call the `Release()` method on an `ICalcPiPtr`, but if you use the ATL wrapper, that would be a compile error.

By default, you get the wrappers when you use `#import`. If you decide that you don't want them, or if for some reason they don't compile (which has been known to happen to at least one of your humble authors on very complex and strange typelibs), you can turn off the wrapper classes and just get straight interface definitions by using the `raw_interfaces_only` option.

ATL Server Web Projects

Without a doubt, the most dramatic recent addition to the ATL library is a suite of classes and tools collectively termed ATL Server. ATL Server accounts for nearly all of the fourfold increase in the overall size of ATL from ATL 3. This extensive class library provides comprehensive support for building web applications and XML web services. Although traditional ASP and the ASP.NET platform offer compelling and easy-to-use frameworks for web-based development, many application developers must still resort to raw ISAPI programming for applications that demand low-level control and maximum performance. ATL Server is designed to provide the performance and control of ISAPI with the feel and productivity of ASP. To that end, ATL Server follows the design model that has made conventional ATL development so effective over the years: namely small, fast, flexible code.

VS provides excellent wizard support for building web applications and web services. Walking through the numerous options available for ATL Server projects is actually quite insightful in understanding both the architecture and the sheer scope of the support provided. VS provides a wizard to help you get started building a web application with ATL Server. You launch this wizard by selecting the ATL Server Project option from the Visual C++ folder of the New Project dialog box.

The Project Settings tab shown in Figure 1.21 displays the selected options for generating and deploying the DLLs that comprise our web application.

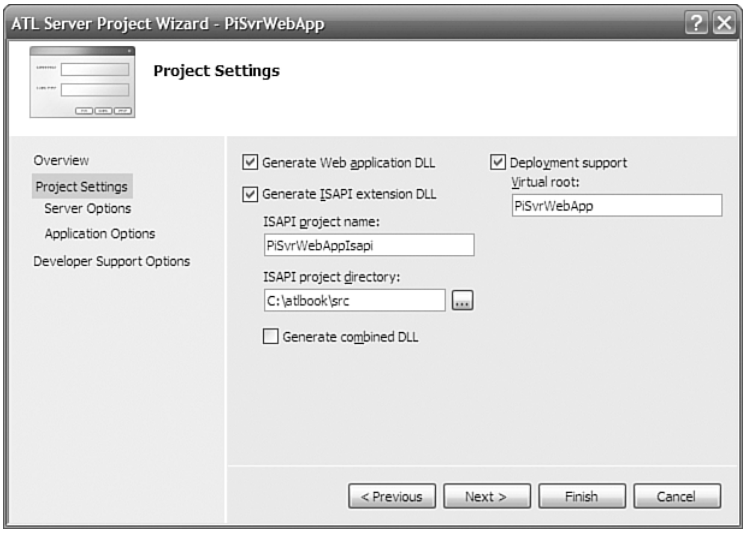

Figure 1.21 Project settings for ATL Server project

By default, ATL Server generates two projects in your solution: a web application DLL and an ISAPI extension DLL. ISAPI extension DLLs are loaded into the IIS process (`inetinfo.exe`) and logically sit between IIS and your web application DLL. Although ISAPI extensions can handle HTTP requests themselves, it is more common for them to provide generic infrastructure services such as thread pooling and caching, leaving web application DLLs to provide the real HTTP response logic. The ATL Server Project Wizard generates an ISAPI extension implementation that communicates with special functions in your web application called handlers. Figure 1.22 depicts this arrangement.

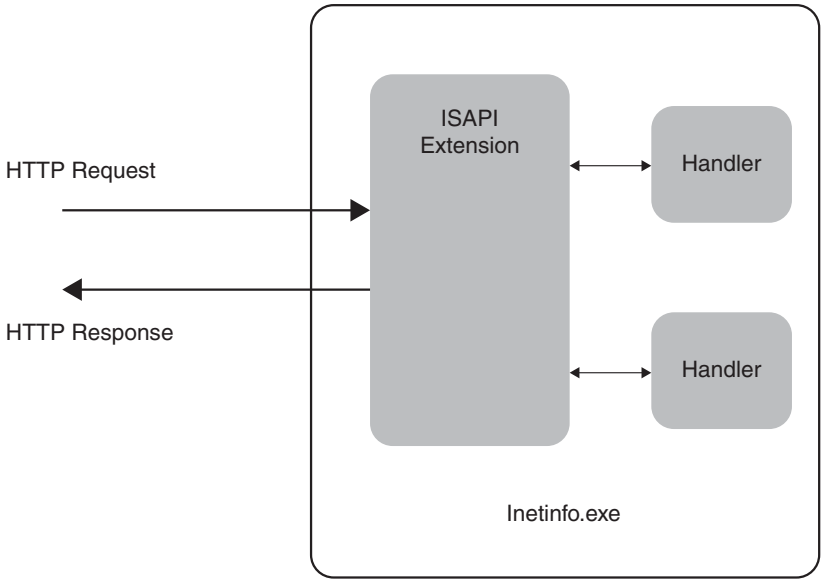

Figure 1.22 Basic ISAPI architecture

The Generate Combined DLL check box enables you to combine everything into a single DLL. This might be an appropriate option if the ISAPI extension is not intended to be used in other web applications. Conversely, developers can opt to leverage ATL Server's extensibility features by creating specialized ISAPI extensions with such options as custom thread pooling, highly tuned caching schemes, or optimized state management. These ISAPI extensions would then likely be reused across multiple web applications. Furthermore, keeping the ISAPI extension as a separate DLL gives us the flexibility to add handlers to our web application without restarting the web server (handler classes are discussed shortly). We'll leave the box unchecked for our first web application and allow VS to generate separate projects.

The Deployment Support check box enables the VS web-deployment tool. With this option selected, the Visual Studio build process automatically performs additional steps for properly deploying your web application so that it is served by IIS. You'll see in a moment how convenient these integrated deployment features can be. A brief word of caution at this point is in order, however. The default setting to enable deployment support causes VS to deploy the built project files in a subdirectory of your default web site, typically `<drive>:\inetpub\wwwroot`. In a real-world development scenario, it might be more desirable to deploy in a different directory on the machine (such as the project directory). Several steps are required to accomplish this, so for now, we're sticking with the default setting just so that we can focus on developing our application.

The Server Options tab shown in Figure 1.23 enables you to select various performance-oriented options for your web application. Several types of caching are supported, including support for arbitrary binary data (Blob cache), file caching, and database connection caching (Data source cache). Additionally, high-availability sites rely upon robust session-state management. ATL Server provides two mechanisms for persisting session state. The OLE DB-backed session-state services radio button includes support for persisting session state in a database (or other OLE DB data source), which is an option suited to applications running on web farms.

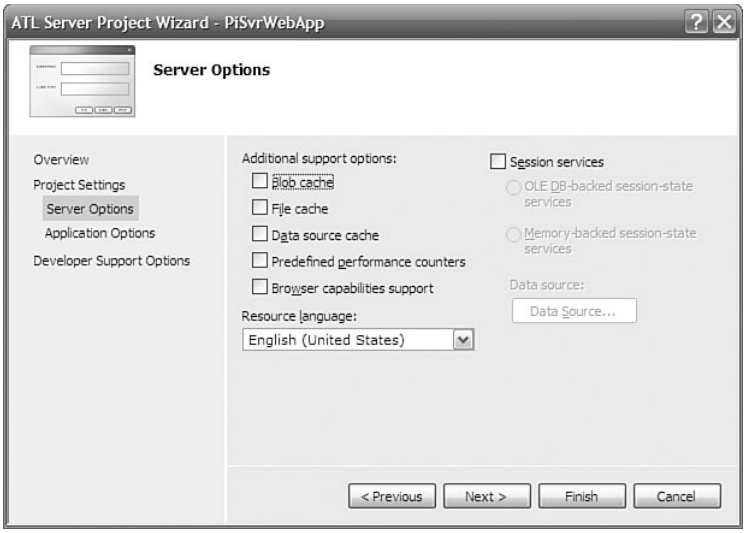

Figure 1.23 Server Options tab for ATL Server project

Figure 1.24 shows the selections available under the Application Options tab. Validation Support generates the necessary code for validating items in the HTTP request from the client, such as query parameters and form variables. Stencil Processing Support generates skeleton code for using HTML code templates known as server response files (SRF). These text files (also known as stencils) end with an `.srf` extension and intermix static HTML content with special replacement tags that your code processes to generate dynamic content at runtime. With stencil processing enabled, the wizard also allows you to select the locale and codepage for properly localizing responses. This simply inserts locale and codepage tags into the generated SRF file. (More on using SRF files comes shortly.) The Create as Web Service option also is discussed further in the following section. Because we're developing a web application, we leave this box unchecked for now.

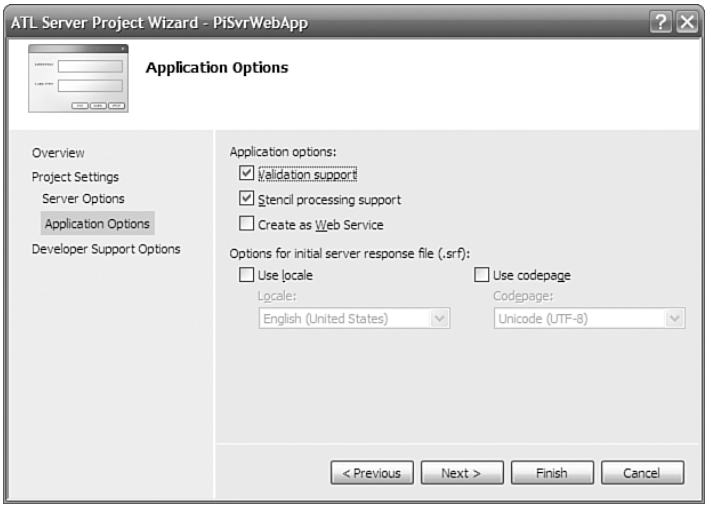

Figure 1.24 ATL Server Application Options tab

The remaining set of options for your ATL Server project appears under the Developer Support Options tab, shown in Figure 1.25. Generating TODO comments simply helps alert the developer to regions of code where additional implementation should be provided. If you select Custom Assert and Trace Handling Support, debug builds of your project will include an instance of the `CDebugReportHook` class, which can greatly simplify the process of debugging your web application—even from a remote machine.

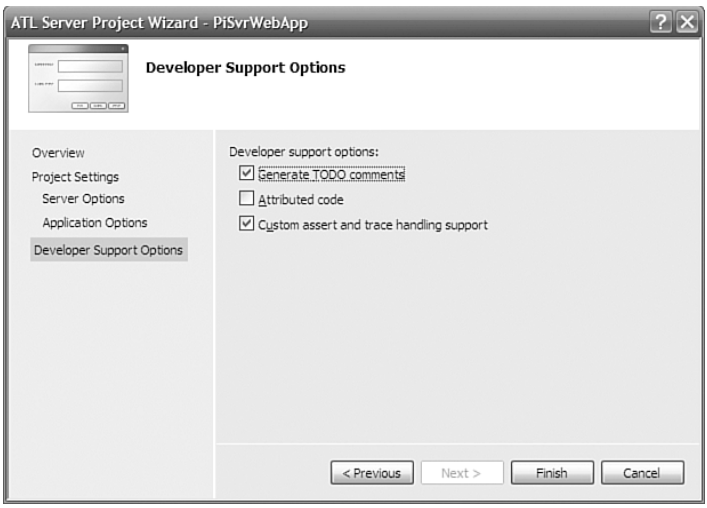

Figure 1.25 ATL Server Developer Support Options tab

Pressing Finish causes the wizard to generate a solution that contains two projects: one for your web application DLL (with a name matching the <projectname> entered in the New Project dialog box) and one for your ISAPI extension (with a name <projectname>Isapi). Let's take a look at the code generated in the ISAPI extension project. The generated .cpp file for our ISAPI extension looks like the following:

```
class CPiSvrWebAppModule :
public CAtlDllModuleT<CPiSvrWebAppModule> {
public:
};

CPiSvrWebAppModule _AtlModule;

typedef CIsapiExtension<> ExtensionType;

// The ATL Server ISAPI extension
ExtensionType theExtension;

// Delegate ISAPI exports to theExtension
//
extern "C"
DWORD WINAPI HttpExtensionProc(LPEXTENSION_CONTROL_BLOCK lpECB) {
    return theExtension.HttpExtensionProc(lpECB);
}

extern "C"
BOOL WINAPI GetExtensionVersion(HSE_VERSION_INFO* pVer) {
    return theExtension.GetExtensionVersion(pVer);
}

extern "C" BOOL WINAPI TerminateExtension(DWORD dwFlags) {
    return theExtension.TerminateExtension(dwFlags);
}

// DLL Entry Point
//
extern "C"
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason,
    LPVOID lpReserved) {
    hInstance;
    return _AtlModule.DllMain(dwReason, lpReserved);
}
```

Because the ISAPI extension uses the services of ATL for object creation, it needs an ATL Module object. Also included in the generated code are implementations of the three well-known entry points IIS uses to communicate HTTP request information to the ISAPI extension: `HttpExtensionProc`, `GetExtensionVersion`, and `TerminateExtension`. These implementations simply delegate to a global instance of `CIsapiExtension`, whose definition is given here:

```
template <
  class ThreadPoolClass=CThreadPool<CIsapiWorker>,
  class CRequestStatClass=CNoRequestStats,
  class HttpUserErrorTextProvider=CDefaultErrorProvider,
  class WorkerThreadTraits=DefaultThreadTraits,
  class CPageCacheStats=CNoStatClass,
  class CStencilCacheStats=CNoStatClass
>
class CIsapiExtension :
  public IServiceProvider,
  public IIsapiExtension,
  public IRequestStats
{ ... }
```

This class provides boilerplate functionality for implementing the ISAPI extension. The template parameters to this class provide pluggable implementation for things such as threadpool management, error reporting, and caching statistics. By replacing this class in the `.cpp` file with your own `CIsapiExtension`-derived class and providing your own classes as template parameters, you can highly customize the behavior of your ISAPI extension. Techniques for doing this are presented in Chapter 13, "Hello, ATL Server." The default implementation of the ISAPI extension is suitable for our demonstration purposes here.

Most of the action takes place in the web application project. The wizard generated a skeleton SRF file for us and placed it in the project. The HTML editor integrated into VS provides a convenient means of viewing and manipulating the contents of this file.

```
<html>
{{ handler PiSvrWebApp.dll/Default }}
    <head>
    </head>
    <body>
        This is a test:  {{Hello}}<br>
    </body>
</html>
```

Items that appear within double braces indicate commands that are passed to the stencil processor. The {{handler}} command specifies the name of the DLL that houses our handler classes for processing replacement tags that appear in the SRF file. The /Default specifier identifies the default request-handler class to use for processing replacement tags. In general, an application DLL can contain multiple handler classes for processing SRF commands, and these classes can even exist in multiple DLLs. We use only a single handler class in a single application DLL, so all commands destined for handler classes will be routed to the same handler class. In the earlier wizard-generated skeleton, the {{Hello}} tag will be passed on to a handler class and replaced by the HTML produced from that class's replacement method.

ATL Server uses several macros to map commands in the SRF file to handler classes in our application DLL. The class definition generated for us in the <projectname>.h file shows how these macros are used:

```
class CPiSvrWebAppHandler
    : public CRequestHandlerT<CPiSvrWebAppHandler>
{
public:
    BEGIN_REPLACEMENT_METHOD_MAP(CPiSvrWebAppHandler)
        REPLACEMENT_METHOD_ENTRY("Hello", OnHello)
    END_REPLACEMENT_METHOD_MAP()

    HTTP_CODE ValidateAndExchange() {
        // Set the content-type
        m_HttpResponse.SetContentType("text/html");
        return HTTP_SUCCESS;
    }

protected:
    HTTP_CODE OnHello(void) {
        m_HttpResponse << "Hello World!";
        return HTTP_SUCCESS;
    }
};
```

The CRequestHandlerT base class provides the implementation for a request-handler class. It uses the REPLACEMENT_METHOD_MAP to map the strings in replacements in the SRF file to the appropriate functions in the class.

In addition to the request-handler class itself, in the handler DLL's .cpp file, you'll find this additional global map:

```
BEGIN_HANDLER_MAP()
    HANDLER_ENTRY("Default", CPiSvrWebAppHandler)
END_HANDLER_MAP()
```

The HANDLER_MAP is used to determine which class to use to process substitutions given with a particular name. In this case, the string "Default" as used in the handler tag in the SRF file is mapped to the CPiSvrWebAppHandler class. When the {{Hello}} tag is encountered in the SRF file, the OnHello method is invoked (via the REPLACEMENT_METHOD_MAP). It uses an instance of CHttpResponse declared as a member variable of the CRequestHandlerT to generate replacement text for the tag.

Let's modify the wizard-generated code to display pi to the number of digits specified in the query string of the HTTP request. First, we modify the SRF file to the following:

```
<html>
{{ handler PiSvrWebApp.dll/Default }}
    <head>
    </head>
    <body>
        PI = {{Pi}}<br>
    </body>
</html>
```

We then add a replacement method called OnPi to our existing handler class and apply the [tag_name] attribute to associate this method with the {{Pi}} replacement tag. In the implementation of the OnPi method, we retrieve the number of digits requested from the query string. The CHttpRequest class stored in m_HttpRequest member variable exposes an instance of CHttpRequestParams. This class provides a simple Lookup method to retrieve individual query parameters from the query string as name-value pairs, so processing requests such as the following is a simple matter:

```
http://localhost/PiSvrWebApp/PiSvrWebApp.srf?digits=6
```

The OnPi method implementation to field such requests follows:

```
class CPiSvrWebAppHandler {
...
    HTTP_CODE OnPi(void) {
        LPCSTR pszDigits = m_HttpRequest.m_QueryParams.Lookup("digits");
        long nDigits = 0;
        if (pszDigits)
```

```
            nDigits = atoi(pszDigits);
        BSTR bstrPi = NULL;
        CalcPi(nDigits, &bstrPi);

        m_HttpResponse << CW2A(bstrPi);
        return HTTP_SUCCESS;
    }
...
};
```

When we build our solution, VS performs a number of convenient tasks on our behalf. Because this is a web application, simply compiling the code into DLLs doesn't quite do the trick. The application must be properly deployed on our web server and registered with IIS. This involves creating a virtual directory, specifying an appropriate level of process isolation, and mapping the `.srf` file extension to our ISAPI extension DLL. Recall that when we created the project, we chose to include deployment support on the Project Settings tab of the ATL Server Project Wizard, shown previously in Figure 1.25. As a result, VS invokes the VCDeploy.exe utility to automatically perform all the necessary web-deployment steps for us. Simply compiling our solution in the normal manner places our application DLL, our ISAPI extension DLL, and our SRF file in a directory under our default web site, typically ending up in the directory `<drive>:\inetpub\wwwroot\<projectName>`. VS uses our web application project name as the virtual directory name, so browsing to `http://localhost/PiSvrWebApp/PiSvrWebApp.srf?digits=50` produces the result in Figure 1.26.

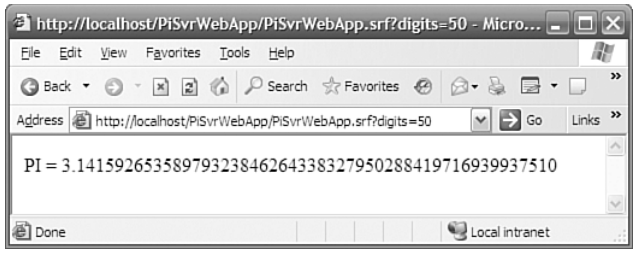

Figure 1.26 Web application for displaying pi to 50 digits

For more information about building ISAPI applications, including web services, with ATL Server, see Chapter 13, "Hello, ATL Server."

Summary

This chapter has been a whirlwind tour through some of the functionality of ATL that the wizards expose, as well as some of the basic interface implementations of ATL. Even with the wizards, it should be clear that ATL is no substitute for solid COM knowledge. You still need to know how to design and implement your interfaces. As you'll see throughout the rest of this book, you still have to know about interface pointers, reference counting, runtime type discovery, threading, persistence . . . the list goes on. ATL can help, but you still need to know COM.

It should also be clear that the wizard is not a substitute for intimate knowledge of ATL or web application development. For every tidbit of ATL information shown in this chapter, there are 10 more salient details, extensions, and pitfalls. And although the wizard saves you typing, it can't do everything. It can't make sure your design and implementation goals are met: That's up to you.

CHAPTER

2 Strings and Text

Strings come in a number of different character sets. COM components often need to use multiple character sets and occasionally need to convert from one set to another. ATL provides a number of string conversion classes that convert from one character set to another, if necessary, and do nothing when they are not needed.

The `CComBSTR` class is a smart string class. This class properly allocates, copies, and frees a string according to the `BSTR` string semantics. `CComBSTR` instances can be used in most, but not all, of the places you would use a `BSTR`.

The `CString` class is a new addition to ATL, with roots in MFC. This class handles allocation, copying, formatting, and offers a host of advanced string-processing features. It can manage ANSI and Unicode data, and convert strings to and from `BSTR`s for use in processing Automation method parameters. With `CString`, you can even control and customize the way memory is managed for the class's string data.

String Data Types, Conversion Classes, and Helper Functions

A Review of Text Data Types

The text data type is somewhat of a pain to deal with in C++ programming. The main problem is that there isn't just one text data type; there are many of them. I use the term *text data type* here in the general sense of an array of characters. Often, different operating systems and programming languages introduce additional semantics for an array of characters (for example, `NUL` character termination or a length prefix) before they consider an array of characters a text string.

When you select a text data type, you must make a number of decisions. First, you must decide what type of characters constitute the array. Some operating systems require you to use ANSI characters when you pass a string (such as a file name) to the operating system. Some operating systems prefer that you use Unicode characters but will accept ANSI characters. Other operating systems require you to use EBCDIC characters. Stranger character sets are in use as well, such as the Multi/Double Byte Character Sets (MBCS/DBCS); this book largely doesn't discuss those details.

Second, you must consider what character set you want to use to manipulate text within your program. No requirement states that your source code must use the same character set that the operating system running your program prefers. Clearly, it's more convenient when both use the same character set, but a program and the operating system can use different character sets. You "simply" must convert all text strings going to and coming from the operating system.

Third, you must determine the length of a text string. Some languages, such as C and C++, and some operating systems, such as Windows 9*x*/NT/XP and UNIX, use a terminating `NUL` character to delimit the end of a text string. Other languages, such as the Microsoft Visual Basic interpreter, Microsoft Java virtual machine, and Pascal, prefer an explicit length prefix specifying the number of characters in the text string.

Finally, in practice, a text string presents a resource-management issue. Text strings typically vary in length. This makes it difficult to allocate memory for the string on the stack—and the text string might not fit on the stack at all. Therefore, text strings are often dynamically allocated. Of course, this means that a text string must be freed eventually. Resource management introduces the idea of an owner of a text string. Only the owner frees the string—and frees it only once. Ownership becomes quite important when you pass a text string between components.

To make matters worse, two COM objects can reside on two different computers running two different operating systems that prefer two different character sets for a text string. For example, you can write one COM object in Visual Basic and run it on the Windows XP operating system. You might pass a text string to another COM object written in C++ running on an IBM mainframe. Clearly, we need some standard text data type that all COM objects in a heterogeneous environment can understand.

COM uses the `OLECHAR` character data type. A COM text string is a `NUL`-character-terminated array of `OLECHAR` characters; a pointer to such a string is an `LPOLESTR`.[1] As a rule, a text string parameter to a COM interface method should be of type `LPOLESTR`. When a method doesn't change the string, the parameter should be of type `LPCOLESTR`—that is, a constant pointer to an array of `OLECHAR` characters.

Frequently, though not always, the `OLECHAR` type isn't the same as the characters you use when writing your code. Sometimes, though not always, the `OLECHAR` type isn't the same as the characters you must provide when passing a text string to the operating system. This means that, depending on context, sometimes you

[1] Note that the actual underlying character data type for `OLECHAR` on one operating system can be different from the underlying character data type for `OLECHAR` on a different operating system. The COM remoting infrastructure performs any necessary character set conversion during marshaling and unmarshaling. Therefore, a COM component always receives text in its expected `OLECHAR` format.

need to convert a text string from one character set to another—and sometimes you won't.

Unfortunately, a change in compiler options (for example, a Windows XP Unicode build or a Windows CE build) can change this context. As a result, code that previously didn't need to convert a string might require conversion, or vice versa. You don't want to rewrite all string-manipulation code each time you change a compiler option. Therefore, ATL provides a number of string-conversion macros that convert a text string from one character set to another and are sensitive to the context in which you invoke the conversion.

Windows Character Data Types

Now let's focus specifically on the Windows platform. Windows-based COM components typically use a mix of four text data types:

- **Unicode.** A specification for representing a character as a "wide-character," 16-bit multilingual character code. The Windows NT/XP operating system uses the Unicode character set internally. All characters used in modern computing worldwide, including technical symbols and special publishing characters, can be represented uniquely in Unicode. The fixed character size simplifies programming when using international character sets. In C/C++, you represent a wide-character string as a `wchar_t` array; a pointer to such a string is a `wchar_t*`.
- **MBCS/DBCS.** The Multi-Byte Character Set is a mixed-width character set in which some characters consist of more than 1 byte. The Windows 9*x* operating systems, in general, use the MBCS to represent characters. The Double-Byte Character Set (DBCS) is a specific type of multibyte character set. It includes some characters that consist of 1 byte and some characters that consist of 2 bytes to represent the symbols for one specific locale, such as the Japanese, Chinese, and Korean languages.

  In C/C++, you represent an MBCS/DBCS string as an `unsigned char` array; a pointer to such a string is an `unsigned char*`. Sometimes a character is one `unsigned char` in length; sometimes, it's more than one. This is loads of fun to deal with, especially when you're trying to back up through a string. In Visual C++, MBCS always means DBCS. Character sets wider than 2 bytes are not supported.
- **ANSI.** You can represent all characters in the English language, as well as many Western European languages, using only 8 bits. Versions of Windows that support such languages use a degenerate case of MBCS, called the Microsoft Windows ANSI character set, in which no multibyte characters are present.

The Microsoft Windows ANSI character set, which is essentially ISO 8859/*x* plus additional characters, was originally based on an ANSI draft standard.

The ANSI character set maps the letters and numerals in the same manner as ASCII. However, ANSI does not support control characters and maps many symbols, including accented letters, that are not mapped in standard ASCII. All Windows fonts are defined in the ANSI character set. This is also called the Single-Byte Character Set (SBCS), for symmetry.

In C/C++, you represent an ANSI string as a `char` array; a pointer to such a string is a `char*`. A character is always one `char` in length. By default, a `char` is a `signed char` in Visual C++. Because MBCS characters are `unsigned` and ANSI characters are, by default, `signed` characters, expressions can evaluate differently when using ANSI characters, compared to using MBCS characters.

- **`TCHAR`/`_TCHAR`.** This is a Microsoft-specific generic-text data type that you can map to a Unicode character, an MBCS character, or an ANSI character using compile-time options. You use this character type to write generic code that can be compiled for any of the three character sets. This simplifies code development for international markets. The C runtime library defines the `_TCHAR` type, and the Windows operating system defines the `TCHAR` type; they are synonymous.

  `tchar.h`, a Microsoft-specific C runtime library header file, defines the generic-text data type `_TCHAR`. ANSI C/C++ compiler compliance requires implementer-defined names to be prefixed by an underscore. When you do not define the `__STDC__` preprocessor symbol (by default, this macro is not defined in Visual C++), you indicate that you don't require ANSI compliance. In this case, the `tchar.h` header file also defines the symbol `TCHAR` as another alias for the generic-text data type if it isn't already defined. `winnt.h`, a Microsoft-specific Win32 operating system header file, defines the generic-text data type `TCHAR`. This header file is operating system specific, so the symbol names don't need the underscore prefix.

Win32 APIs and Strings

Each Win32 API that requires a string has two versions: one that requires a Unicode argument and another that requires an MBCS argument. On a non-MBCS-enabled version of Windows, the MBCS version of an API expects an ANSI argument. For example, the `SetWindowText` API doesn't really exist. There are actually two functions: `SetWindowTextW`, which expects a Unicode string argument, and `SetWindowTextA`, which expects an MBCS/ANSI string argument.

The Windows NT/2000/XP operating systems internally use only Unicode strings. Therefore, when you call `SetWindowTextA` on Windows NT/2000/XP, the

function translates the specified string to Unicode and then calls `SetWindowTextW`. The Windows 9*x* operating systems do not support Unicode directly. The `SetWindowTextA` function on the Windows 9*x* operating systems does the work, while `SetWindowTextW` returns an error. The MSLU library from Microsoft[2] provides implementations of almost all the Unicode functions on Win9*x*.

This gives you a difficult choice. You could write a performance-optimized component using Unicode character strings that runs on Windows 2000 but not on Windows 9*x*. You could use MSLU for Unicode strings on both families and lose performance on Windows 9*x*. You could write a more general component using MBCS/ANSI character strings that runs on both operating systems but *not* optimally on Windows 2000. Alternatively, you could hedge your bets by writing source code that enables you to decide at compile time what character set to support.

A little coding discipline and some preprocessor magic let you code as if there were a single API called `SetWindowText` that expects a `TCHAR` string argument. You specify at compile time which kind of component you want to build. For example, you write code that calls `SetWindowText` and specifies a `TCHAR` buffer. When compiling a component as Unicode, you call `SetWindowTextW`; the argument is a `wchar_t` buffer. When compiling an MBCS/ANSI component, you call `SetWindowTextA`; the argument is a `char` buffer.

When you write a Windows-based COM component, you should typically use the `TCHAR` character type to represent characters used by the component internally. Additionally, you should use it for all characters used in interactions with the operating system. Similarly, you should use the `TEXT` or `__TEXT` macro to surround every literal character or string.

`tchar.h` defines the functionally equivalent macros `_T` , `__T`, and `_TEXT`, which all compile a character or string literal as a generic-text character or literal. `winnt.h` also defines the functionally equivalent macros `TEXT` and `__TEXT`, which are yet more synonyms for `_T`, `__T`, and `_TEXT`. (There's nothing like five ways to do exactly the same thing.) The examples in this chapter use `__TEXT` because it's defined in `winnt.h`. I actually prefer `_T` because it's less clutter in my source code.

An operating-system-agnostic coding approach favors including `tchar.h` and using the `_TCHAR` generic-text data type because that's somewhat less tied to the Windows operating systems. However, we're discussing building components with text handling optimized at compile time for specific versions of the Windows operating systems. This argues that we should use `TCHAR`, the type defined in `winnt.h`. Plus, `TCHAR` isn't as jarring to the eyes as `_TCHAR` and it's easier to type. Most code already implicitly includes the `winnt.h` header file via `windows.h`, and you must

[2] More information on MSLU is available at http://www.microsoft.com/globaldev/handson/dev/mslu_announce.mspx (http://tinysells.com/49).

explicitly include tchar.h. All sorts of good reasons support using TCHAR, so the examples in this book use this as the generic-text data type.

This means that you can compile specialized versions of the component for different markets or for performance reasons. These types and macros are defined in the winnt.h header file.

You also must use a different set of string runtime library functions when manipulating strings of TCHAR characters. The familiar functions strlen, strcpy, and so on operate only on char characters. The less familiar functions wcslen, wcscpy, and so on work on wchar_t characters. Moreover, the totally strange functions _mbslen, _mbscpy, and so on work on multibyte characters. Because TCHAR characters are sometimes wchar_t, sometimes char-holding ANSI characters, and sometimes char-holding (nominally unsigned) multibyte characters, you need an equivalent set of runtime library functions that work with TCHAR characters.

The tchar.h header file defines a number of useful generic-text mappings for string-handling functions. These functions expect TCHAR parameters, so all their function names use the _tcs (the _t character set) prefix. For example, _tcslen is equivalent to the C runtime library strlen function. The _tcslen function expects TCHAR characters, whereas the strlen function expects char characters.

Controlling Generic-Text Mapping Using the Preprocessor

Two preprocessor symbols and two macros control the mapping of the TCHAR data type to the underlying character type the application uses.

- **UNICODE/_UNICODE.** The header files for the Windows operating system APIs use the UNICODE preprocessor symbol. The C/C++ runtime library header files use the _UNICODE preprocessor symbol. Typically, you define either both symbols or neither of them. When you compile with the symbol _UNICODE defined, tchar.h maps all TCHAR characters to wchar_t characters. The _T, __T, and _TEXT macros prefix each character or string literal with a capital L (creating a Unicode character or literal, respectively). When you compile with the symbol UNICODE defined, winnt.h maps all TCHAR characters to wchar_t characters. The TEXT and __TEXT macros prefix each character or string literal with a capital L (creating a Unicode character or literal, respectively). The _tcsXXX functions are mapped to the corresponding _wcsXXX functions.
- **_MBCS.** When you compile with the symbol _MBCS defined, all TCHAR characters map to char characters, and the preprocessor removes all the _T and __TEXT macro variations. It leaves the character or literal unchanged (creating an MBCS character or literal, respectively). The _tcsXXX functions are mapped to the corresponding _mbsXXX versions.

- **None of the above.** When you compile with neither symbol defined, all TCHAR characters map to char characters and the preprocessor removes all the _T and __TEXT macro variations, leaving the character or literal unchanged (creating an ANSI character or literal, respectively). The _tcsXXX functions are mapped to the corresponding strXXX functions.

You write generic-text-compatible code by using the generic-text data types and functions. An example of reversing and concatenating to a generic-text string follows:

```
TCHAR *reversedString, *sourceString, *completeString;
reversedString = _tcsrev (sourceString);
completeString = _tcscat (reversedString, __TEXT("suffix"));
```

When you compile the code without defining any preprocessor symbols, the preprocessor produces this output:

```
char *reversedString, *sourceString, *completeString;
reversedString = _strrev (sourceString);
completeString = strcat (reversedString, "suffix");
```

When you compile the code after defining the _UNICODE preprocessor symbol, the preprocessor produces this output:

```
wchar_t *reversedString, *sourceString, *completeString;
reversedString = _wcsrev (sourceString);
completeString = wcscat (reversedString, L"suffix");
```

When you compile the code after defining the _MBCS preprocessor symbol, the preprocessor produces this output:

```
char *reversedString, *sourceString, *completeString;
reversedString = _mbsrev (sourceString);
completeString = _mbscat (reversedString, "suffix");
```

COM Character Data Types

COM uses two character types:

- **OLECHAR.** The character type COM uses on the operating system for which you compile your source code. For Win32 operating systems, this is the wchar_t

character type.[3] For Win16 operating systems, this is the `char` character type. For the Mac OS, this is the `char` character type. For the Solaris OS, this is the `wchar_t` character type. For the as yet unknown operating system, this is who knows what. Let's just pretend there is an abstract data type called `OLECHAR`. COM uses it. Don't rely on it mapping to any specific underlying data type.

- **`BSTR`.** A specialized string type some COM components use. A `BSTR` is a length-prefixed array of `OLECHAR` characters with numerous special semantics.

Now let's complicate things a bit. You want to write code for which you can select, at compile time, the type of characters it uses. Therefore, you're manipulating strictly `TCHAR` strings internally. You also want to call a COM method and pass it the same strings. You must pass the method either an `OLECHAR` string or a `BSTR` string, depending on its signature. The strings your component uses might or might not be in the correct character format, depending on your compilation options. This is a job for Supermacro!

ATL String-Conversion Classes

ATL provides a number of string-conversion classes that convert, when necessary, among the various character types described previously. The classes perform no conversion and, in fact, do nothing, when the compilation options make the source and destination character types identical. Seven different classes in `atlconv.h` implement the real conversion logic, but this header also uses a number of typedefs and preprocessor `#define` statements to make using these converter classes syntactically more convenient.

These class names use a number of abbreviations for the various character data types:

- **T** represents a pointer to the Win32 `TCHAR` character type—an `LPTSTR` parameter.
- **W** represents a pointer to the Unicode `wchar_t` character type—an `LPWSTR` parameter.
- **A** represents a pointer to the MBCS/ANSI `char` character type—an `LPSTR` parameter.
- **OLE** represents a pointer to the COM `OLECHAR` character type—an `LPOLESTR` parameter.
- **C** represents the C/C++ `const` modifier.

[3] Actually, you can change the Win32 `OLECHAR` data type from the default `wchar_t` (which COM uses internally) to char by defining the preprocessor symbol `OLE2ANSI`. This lets you pretend that COM uses ANSI. MFC once used this feature, but it no longer does and neither should you.

All class names use the form C<source-abbreviation>2<destination-abbreviation>. For example, the CA2W class converts an LPSTR to an LPWSTR. When there is a C in the name (not including the first C—that stands for "class"), add a const modification to the following abbreviation; for example, the CT2CW class converts a LPTSTR to a LPCWSTR.

The actual class behavior depends on which preprocessor symbols you define (see Table 2.1). Note that the ATL conversion classes and macros treat OLE and W as equivalent.

Table 2.1. Character Set Preprocessor Symbols

Preprocessor Symbol Defined	T Becomes . . .	OLE Becomes . . .
None	A	W
_UNICODE	W	W

Table 2.2 lists the ATL string-conversion macros.

Table 2.2. ATL String-Conversion Classes

CA2W	COLE2T	CT2CA	CT2W	CW2T
CA2WEX	COLE2TEX	CT2CAEX	CT2WEX	CW2TEX
CA2T	COLE2CT	CT2OLE	CT2CW	CW2CT
CA2TEX	COLE2CTEX	CT2OLEEX	CT2CWEX	CW2CTEX
CA2CT	CT2A	CT2COLE	CW2A	
CA2CTEX	CT2AEX	CT2COLEEX	CW2AEX	

As you can see, no BSTR conversion classes are listed in Table 2.2. The next section of this chapter introduces the CComBSTR class as the preferred mechanism for dealing with BSTR-type conversions.

When you look inside the atlconv.h header file, you'll see that many of the definitions distill down to a fairly small set of six actual classes. For instance, when _UNICODE is defined, CT2A becomes CW2A, which is itself typedef'd to the CW2AEX template class. The type definition merely applies the default template parameters to CW2AEX. Additionally, all the previous class names always map OLE to W, so COLE2T

becomes CW2T, which is defined as CW2W under Unicode builds. Because the source and destination types for CW2W are the same, this class performs no conversions. Ultimately, the only six classes defined are the template classes CA2AEX, CA2CAEX, CA2WEX, CW2AEX, CW2CWEX, and CW2WEX. Only CA2WEX and CW2AEX have different source and destination types, so these are the only two classes doing any real work. Thus, our expansive list of conversion classes in Table 2.2 has distilled down to only two interesting ones. These two classes are both defined and implemented similarly, so we look at only CA2WEX to glean an understanding of how they both work.

```
template< int t_nBufferLength = 128 >
class CA2WEX {
    CA2WEX( LPCSTR psz );
    CA2WEX( LPCSTR psz, UINT nCodePage );
    ...
public:
    LPWSTR m_psz;
    wchar_t m_szBuffer[t_nBufferLength];
    ...
};
```

The class definition is actually pretty simple. The template parameter specifies the size of a fixed static buffer to hold the string data. This means that most string-conversion operations can be performed without allocating any dynamic storage. If the requested string to convert exceeds the number of characters passed as an argument to the template, CA2WEX uses malloc to allocate additional storage.

Two constructors are provided for CA2WEX. The first constructor accepts an LPCSTR and uses the Win32 API function MultiByteToWideChar to perform the conversion. By default, the class uses the ANSI code page for the current thread's locale to perform the conversion. The second constructor can be used to specify an alternate code page that governs how the conversion is performed. This value is passed directly to MultiByteToWideChar, so see the online documentation for details on code pages accepted by the various Win32 character conversion functions.

The simplest way to use this converter class is to accept the default value for the buffer size parameter. Thus, ATL provides a simple typedef to facilitate this:

```
typedef CA2WEX<> CA2W;
```

To use this converter class, you need to write only simple code such as the following:

```
void PutName (LPCWSTR lpwszName);

void RegisterName (LPCSTR lpsz) {
    PutName (CA2W(lpsz));
}
```

Two other use cases are also common in practice:

1. Receiving a generic-text string and passing to a method that expects an `OLESTR` as input
2. Receiving an `OLESTR` and passing it to a method that expects a generic-text string

The conversion classes are easily employed to deal with these cases:

```
void PutAddress(LPOLESTR lpszAddress);

void RegisterAddress(LPTSTR lpsz) {
    PutAddress(CT2OLE(lpsz));
}

void PutNickName(LPTSTR lpszName);

void RegisterAddress(LPOLESTR lpsz) {
    PutNickName(COLE2T(lpsz));
}
```

A Note on Memory Management

As convenient as the conversion classes are, you can run into some nasty pitfalls if you use them incorrectly. The conversion classes allocate the memory for the converted text automatically and clean it up in the class destructor. This is useful because you don't have to worry about buffer management. However, it also means that code like this is a crash waiting to happen:

```
LPOLESTR ConvertString(LPTSTR lpsz) {
    return CT2OLE(lpsz);
}
```

You've just returned either a pointer to the stack of the called function (which is trashed when the function returns) if the string was short, or a pointer to an array on the heap that will be deallocated before the function returns.

The worst part is that, depending on your macro selection, the code might work just fine but will crash when you switch from ANSI to Unicode for the first time (usually two days before ship). To avoid this, make sure that you copy the converted string to a separate buffer (or use a string class) first if you need it for more than a single expression.

ATL String-Helper Functions

Sometimes you want to copy a string of `OLECHAR` characters. You also happen to know that `OLECHAR` characters are wide characters on the Win32 operating system. When writing a Win32 version of your component, you might call the Win32 operating system function `lstrcpyW`, which copies wide characters. Unfortunately, Windows NT/2000, which supports Unicode, implements `lstrcpyW`, but Windows 95 does not. A component that uses the `lstrcpyW` API doesn't work correctly on Windows 95.

Instead of `lstrcpyW`, use the ATL string-helper function `ocscpy` to copy an `OLECHAR` character string. It works properly on both Windows NT/2000 and Windows 95. The ATL string-helper function `ocslen` returns the length of an `OLECHAR` string. This is nice for symmetry, although the `lstrlenW` function it replaces does work on both operating systems.

```
OLECHAR* ocscpy(LPOLESTR dest, LPCOLESTR src);
size_t ocslen(LPCOLESTR s);
```

Similarly, the Win32 `CharNextW` operating system function doesn't work on Windows 95, so ATL provides a `CharNextO` string-helper function that increments an `OLECHAR*` by one character and returns the next character pointer. It does not increment the pointer beyond a `NUL` termination character.

```
LPOLESTR CharNextO(LPCOLESTR lp);
```

ATL String-Conversion Macros

The string-conversion classes discussed previously were introduced in ATL 7. ATL 3 (and code written with ATL 3) used a set of macros instead. In fact, these macros are still in use in the ATL code base. For example, this code is in the `atlctl.h` header:

```
    STDMETHOD(Help)(LPCOLESTR pszHelpDir) {
        T* pT = static_cast<T*>(this);
        USES_CONVERSION;
```

```
        ATLTRACE(atlTraceControls,2,
          _T("IPropertyPageImpl::Help\n"));
        CComBSTR szFullFileName(pszHelpDir);
        CComHeapPtr<OLECHAR>
          pszFileName(LoadStringHelper(pT->m_dwHelpFileID));
        if (pszFileName == NULL)
          return E_OUTOFMEMORY;
        szFullFileName.Append(OLESTR("\\"));
        szFullFileName.Append(pszFileName);
        WinHelp(pT->m_hWnd, OLE2CT(szFullFileName),
            HELP_CONTEXTPOPUP, NULL);
        return S_OK;
}
```

The macros behave much like the conversion classes, minus the leading `C` in the macro name. So, to convert from `tchar` to `olechar`, you use `T2OLE(s)`.

Two major differences arise between the macros and the conversion classes. First, the macros require some local variables to work; the `USES_CONVERSION` macro is required in any function that uses the conversion macros. (It declares these local variables.) The second difference is the location of the conversion buffer.

In the conversion classes, the buffer is stored either as a member variable on the stack (if the buffer is small) or on the heap (if the buffer is large). The conversion macros always use the stack. They call the runtime function `_alloca`, which allocates extra space on the local stack.

Although it is fast, `_alloca` has some serious downsides. The stack space isn't freed until the function exits, which means that if you do conversion in a loop, you might end up blowing out your stack space. Another nasty problem is that if you use the conversion macros inside a C++ `catch` block, the `_alloca` call messes up the exception-tracking information on the stack and you crash.[4]

The ATL team apparently took two swipes at improving the conversion macros. The final solution is the conversion classes. However, a second set of conversion macros exists: the `_EX` flavor. These are used much like the original conversion macros; you put `USES_CONVERSION_EX` at the top of the function. The macros have an `_EX` suffix, as in `T2A_EX`. The `_EX` macros are different, however: They take two parameters, not one. The first parameter is the buffer to convert from as usual. The second parameter is a threshold value. If the converted buffer is smaller than this threshold, the memory is allocated via `_alloca`. If the buffer is larger, it is allocated on the heap instead. So, these macros give you a chance to avoid the stack overflow.

[4] For this reason, the `_alloca` function is deprecated in favor of `_malloca`, but ATL still uses `_alloca`.

(They still won't help you in a `catch` block.) The ATL code uses the `_EX` macros extensively; the previous example is the only one left that still uses the old macros.

We don't go into the details of either macro set here; the conversion classes are much safer to use and are preferred for new code. We mention them only so that you know what you're looking at if you see them in older code or the ATL sources themselves.

The CComBSTR Smart BSTR Class

A Review of the COM String Data Type: BSTR

COM is a language-neutral, hardware-architecture-neutral model. Therefore, it needs a language-neutral, hardware-architecture-neutral text data type. COM defines a generic text data type, `OLECHAR`, that represents the text data COM uses on a specific platform. On most platforms, including all 32-bit Windows platforms, the `OLECHAR` data type is a typedef for the `wchar_t` data type. That is, on most platforms, the COM text data type is equivalent to the C/C++ wide-character data type, which contains Unicode characters. On some platforms, such as the 16-bit Windows operating system, `OLECHAR` is a `typedef` for the standard C `char` data type, which contains ANSI characters. Generally, you should define all string parameters used in a COM interface as `OLECHAR*` arguments.

COM also defines a text data type called `BSTR`. A `BSTR` is a length-prefixed string of `OLECHAR` characters. Most interpretive environments prefer length-prefixed strings for performance reasons. For example, a length-prefixed string does not require time-consuming scans for a `NUL` character terminator to determine the length of a string. Actually, the `NUL`-character-terminated string is a language-specific concept that was originally unique to the C/C++ language. The Microsoft Visual Basic interpreter, the Microsoft Java virtual machine, and most scripting languages, such as VBScript and JScript, internally represent a string as a `BSTR`.

Therefore, when you pass a string to or receive a string from a method parameter to an interface defined by a C/C++ component, you'll often use the `OLECHAR*` data type. However, if you need to use an interface defined by another language, frequently string parameters will be the `BSTR` data type. The `BSTR` data type has a number of poorly documented semantics, which makes using `BSTRs` tedious and error prone for C++ developers.

A `BSTR` has the following attributes:

- A `BSTR` is a pointer to a length-prefixed array of `OLECHAR` characters.
- A `BSTR` is a pointer data type. It points at the first character in the array. The length prefix is stored as an integer immediately preceding the first character in the array.

- The array of characters is NUL character terminated.
- The length prefix is in bytes, not characters, and does not include the terminating NUL character.
- The array of characters may contain embedded NUL characters.
- A BSTR must be allocated and freed using the SysAllocString and SysFreeString family of functions.
- A NULL BSTR pointer implies an empty string.
- A BSTR is not reference counted; therefore, two references to the same string content must refer to separate BSTRs. In other words, copying a BSTR implies making a duplicate string, not simply copying the pointer.

With all these special semantics, it would be useful to encapsulate these details in a reusable class. ATL provides such a class: CComBSTR.

The CComBSTR Class

The CComBSTR class is an ATL utility class that is a useful encapsulation for the COM string data type, BSTR. The atlcomcli.h file contains the definition of the CComBSTR class. The only state maintained by the class is a single public member variable, m_str, of type BSTR.

```
//////////////////////////////////////////////////
// CComBSTR

class CComBSTR {
public:
  BSTR m_str;
...
} ;
```

Constructors and Destructor

Eight constructors are available for CComBSTR objects. The default constructor simply initializes the m_str variable to NULL, which is equivalent to a BSTR that represents an empty string. The destructor destroys any BSTR contained in the m_str variable by calling SysFreeString. The SysFreeString function explicitly documents that the function simply returns when the input parameter is NULL so that the destructor can run on an empty object without a problem.

```
    CComBSTR() { m_str = NULL; }
    ~CComBSTR() { ::SysFreeString(m_str); }
```

Later in this section, you will learn about numerous convenience methods that the CComBSTR class provides. However, one of the most compelling reasons for using the class is so that the destructor frees the internal BSTR at the appropriate time, so you don't have to free a BSTR explicitly. This is exceptionally convenient during times such as stack frame unwinding when locating an exception handler.

Probably the most frequently used constructor initializes a CComBSTR object from a pointer to a NUL-character-terminated array of OLECHAR characters—or, as it's more commonly known, an LPCOLESTR.

```
CComBSTR(LPCOLESTR pSrc) {
    if (pSrc == NULL) m_str = NULL;
    else {
        m_str = ::SysAllocString(pSrc);
        if (m_str == NULL)
                AtlThrow(E_OUTOFMEMORY);
    }
}
```

You invoke the preceding constructor when you write code such as the following:

```
CComBSTR str1 (OLESTR ("This is a string of OLECHARs")) ;[5]
```

The previous constructor copies characters until it finds the end-of-string NULL character terminator. When you want some lesser number of characters copied, such as the prefix to a string, or when you want to copy from a string that contains embedded NULL characters, you must explicitly specify the number of characters to copy. In this case, use the following constructor:

```
CComBSTR(int nSize, LPCOLESTR sz);
```

This constructor creates a BSTR with room for the number of characters specified by nSize; copies the specified number of characters, including any embedded NULL characters, from sz; and then appends a terminating NUL character. When sz is

[5] The OLESTR macro is similar to the _T macros; it guarantees that the string literal is of the proper type for an OLE string, depending on compile options.

NULL, SysAllocStringLen skips the copy step, creating an uninitialized BSTR of the specified size. You invoke the preceding constructor when you write code such as the following:

```
// str2 contains "This is a string"
CComBSTR str2 (16, OLESTR ("This is a string of OLECHARs"));

// Allocates an uninitialized BSTR with room for 64 characters
CComBSTR str3 (64, (LPCOLESTR) NULL);

// Allocates an uninitialized BSTR with room for 64 characters
CComBSTR str4 (64);
```

The CComBSTR class provides a special constructor for the str3 example in the preceding code, which doesn't require you to provide the NULL argument. The preceding str4 example shows its use. Here's the constructor:

```
CComBSTR(int nSize) {
    ...
    m_str = ::SysAllocStringLen(NULL, nSize);
    ...
}
```

One odd semantic feature of a BSTR is that a NULL pointer is a valid value for an empty BSTR string. For example, Visual Basic considers a NULL BSTR to be equivalent to a pointer to an empty string—that is, a string of zero length in which the first character is the terminating NUL character. To put it symbolically, Visual Basic considers IF p = "", where p is a BSTR set to NULL, to be true. The SysStringLen API properly implements the checks; CComBSTR provides the Length method as a wrapper:

```
unsigned int Length() const { return ::SysStringLen(m_str); }
```

You can also use the following copy constructor to create and initialize a CComBSTR object to be equivalent to an already initialized CComBSTR object:

```
CComBSTR(const CComBSTR& src) {
     m_str = src.Copy();
     ...
}
```

In the following code, creating the str5 variable invokes the preceding copy constructor to initialize their respective objects:

```
CComBSTR str1 (OLESTR("This is a string of OLECHARs")) ;
CComBSTR str5 = str1 ;
```

Note that the preceding copy constructor calls the Copy method on the source CComBSTR object. The Copy method makes a copy of its string and returns the new BSTR. Because the Copy method allocates the new BSTR using the length of the existing BSTR and copies the string contents for the specified length, the Copy method properly copies a BSTR that contains embedded NUL characters.

```
    BSTR Copy() const {
        if (!*this) { return NULL; }
        return ::SysAllocStringByteLen((char*)m_str,
            ::SysStringByteLen(m_str));
    }
```

Two constructors initialize a CComBSTR object from an LPCSTR string. The single argument constructor expects a NUL-terminated LPCSTR string. The two-argument constructor permits you to specify the length of the LPCSTR string. These two constructors are functionally equivalent to the two previously discussed constructors that accept an LPCOLESTR parameter. The following two constructors expect ANSI characters and create a BSTR that contains the equivalent string in OLECHAR characters:

```
CComBSTR(LPCSTR pSrc) {
     ...
     m_str = A2WBSTR(pSrc);
     ...
}
CComBSTR(int nSize, LPCSTR sz) {
     ...
     m_str = A2WBSTR(sz, nSize);
     ...
}
```

The final constructor is an odd one. It takes an argument that is a GUID and produces a string containing the string representation of the GUID.

```
CComBSTR(REFGUID src);
```

This constructor is quite useful when building strings used during component registration. In a number of situations, you need to write the string representation of a GUID to the Registry. Some code that uses this constructor follows:

```
// Define a GUID as a binary constant
static const GUID GUID_Sample = { 0x8a44e110, 0xf134, 0x11d1,
    { 0x96, 0xb1, 0xBA, 0xDB, 0xAD, 0xBA, 0xDB, 0xAD } };

// Convert the binary GUID to its string representation
CComBSTR str6 (GUID_Sample) ;
// str6 contains "{8A44E110-F134-11d1-96B1-BADBADBADBAD}"
```

Assignment

The `CComBSTR` class defines three assignment operators. The first one initializes a `CComBSTR` object using a different `CComBSTR` object. The second one initializes a `CComBSTR` object using an `LPCOLESTR` pointer. The third one initializes the object using a `LPCSTR` pointer. The following `operator=()` method initializes one `CComBSTR` object from another `CComBSTR` object:

```
CComBSTR& operator=(const CComBSTR& src) {
    if (m_str != src.m_str) {
        ::SysFreeString(m_str);
        m_str = src.Copy();
        if (!!src && !*this) { AtlThrow(E_OUTOFMEMORY); }
    }
    return *this;
}
```

Note that this assignment operator uses the `Copy` method, discussed a little later in this section, to make an exact copy of the specified `CComBSTR` instance. You invoke this operator when you write code such as the following:

```
CComBSTR str1 (OLESTR("This is a string of OLECHARs"));
CComBSTR str7 ;

str7 = str1; // str7 contains "This is a string of OLECHARs"
str7 = str7; // This is a NOP. Assignment operator
             // detects this case
```

The second `operator=()` method initializes one `CComBSTR` object from an `LPCOLESTR` pointer to a `NUL`-character-terminated string.

```
CComBSTR& operator=(LPCOLESTR pSrc) {
    if (pSrc != m_str) {
        ::SysFreeString(m_str);
        if (pSrc != NULL) {
            m_str = ::SysAllocString(pSrc);
            if (!*this) { AtlThrow(E_OUTOFMEMORY); }
        } else {
            m_str = NULL;
        }
    }
    return *this;
}
```

Note that this assignment operator uses the `SysAllocString` function to allocate a `BSTR` copy of the specified `LPCOLESTR` argument. You invoke this operator when you write code such as the following:

```
CComBSTR str8 ;

str8 = OLESTR ("This is a string of OLECHARs");
```

It's quite easy to misuse this assignment operator when you're dealing with strings that contain embedded `NUL` characters. For example, the following code demonstrates how to use and misuse this method:

```
CComBSTR str9 ;
str9 = OLESTR ("This works as expected");

// BSTR bstrInput contains "This is part one\0and here's part two"
CComBSTR str10 ;
str10 = bstrInput; // str10 now contains "This is part one"
```

To properly handle situations such as this one, you should turn to the `AssignBSTR` method. This method is implemented very much like `operator=(LPCOLESTR)`, except that it uses `SysAllocStringByteLen`.

```
HRESULT AssignBSTR(const BSTR bstrSrc) {
    HRESULT hr = S_OK;
    if (m_str != bstrSrc) {
        ::SysFreeString(m_str);
        if (bstrSrc != NULL) {
            m_str = ::SysAllocStringByteLen((char*)bstrSrc,
                ::SysStringByteLen(bstrSrc));
```

```
            if (!*this) { hr = E_OUTOFMEMORY; }
        } else {
            m_str = NULL;
        }
    }

    return hr;
}
```

You can modify the code as follows:

```
CComBSTR str9 ;
str9 = OLESTR ("This works as expected");

// BSTR bstrInput contains
// "This is part one\0and here's part two"
CComBSTR str10 ;
str10.AssignBSTR(bstrInput);     // works properly

// str10 now contains "This is part one\0and here's part two"
```

The third `operator=()` method initializes one `CComBSTR` object using an `LPCSTR` pointer to a `NUL`-character-terminated string. The operator converts the input string, which is in ANSI characters, to a Unicode string; then it creates a `BSTR` containing the Unicode string.

```
CComBSTR& operator=(LPCSTR pSrc) {
    ::SysFreeString(m_str);
    m_str = A2WBSTR(pSrc);
    if (!*this && pSrc != NULL) { AtlThrow(E_OUTOFMEMORY); }
    return *this;
}
```

The final assignment methods are two overloaded methods called `LoadString`.

```
bool LoadString(HINSTANCE hInst, UINT nID) ;
bool LoadString(UINT nID) ;
```

The first loads the specified string resource `nID` from the specified module `hInst` (using the instance handle). The second loads the specified string resource `nID` from the current module using the global variable `_AtlBaseModule`.

CComBSTR Operations

Four methods give you access, in varying ways, to the internal BSTR string that is encapsulated by the CComBSTR class. The operator BSTR() method enables you to use a CComBSTR object in situations where a raw BSTR pointer is required. You invoke this method any time you cast a CComBSTR object to a BSTR implicitly or explicitly.

```
operator BSTR() const { return m_str; }
```

Frequently, you invoke this operator implicitly when you pass a CComBSTR object as a parameter to a function that expects a BSTR. The following code demonstrates this:

```
HRESULT put_Name (/* [in] */ BSTR pNewValue) ;

CComBSTR bstrName = OLESTR ("Frodo Baggins");
put_Name (bstrName); // Implicit cast to BSTR
```

The operator&() method returns the address of the internal m_str variable when you take the address of a CComBSTR object. Use care when taking the address of a CComBSTR object. Because the operator&() method returns the address of the internal BSTR variable, you can overwrite the internal variable without first freeing the string. This causes a memory leak. However, if you define the macro ATL_CCOMBSTR_ADDRESS_OF_ASSERT in your project settings, you get an assertion to help catch this error.

```
#ifndef ATL_CCOMBSTR_ADDRESS_OF_ASSERT
// Temp disable CComBSTR::operator& Assert
#define ATL_NO_CCOMBSTR_ADDRESS_OF_ASSERT
#endif

BSTR* operator&() {
#ifndef ATL_NO_CCOMBSTR_ADDRESS_OF_ASSERT
    ATLASSERT(!*this);
#endif
    return &m_str;
}
```

This operator is quite useful when you are receiving a BSTR pointer as the output of some method call. You can store the returned BSTR directly into a CComBSTR object so that the object manages the lifetime of the string.