

Craig McMurtry
Marc Mercuri
Nigel Watling
Matt Winkler

Windows® Communication Foundation 3.5

UNLEASHED



SAMS

Craig McMurtry
Marc Mercuri
Nigel Watling
Matt Winkler

Windows Communication Foundation 3.5

UNLEASHED



800 East 96th Street, Indianapolis, Indiana 46240 USA

Windows Communication Foundation 3.5 Unleashed

Copyright © 2009 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-672-33024-7

ISBN-10: 0-672-33024-5

Library of Congress Cataloging-in-Publication Data:

Windows Communication Foundation 3.5 unleashed / Craig McMurtry ... [et al.]. – 2nd ed.

p. cm.

Includes bibliographical references and index.

ISBN 978-0-672-33024-7

1. Application software–Development. 2. Electronic data processing–Distributed processing. 3. Microsoft Windows (Computer file) 4. Web services. I. McMurtry, Craig.

QA76.76.A65W59 2009

005.4'46–dc22

2008038773

Printed in the United States of America

First Printing October 2008

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Bulk Sales

Pearson offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact:

U.S. Corporate and Government Sales

1-800-382-3419

corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact:

International Sales

+1-317-581-3793

international@pearsontechgroup.com

Editor-in-Chief

Karen Gettman

Executive Editor

Neil Rowe

Acquisitions Editor

Brook Farling

Development Editor

Mark Renfrow

Managing Editor

Patrick Kanouse

Project Editor

SanDee Phillips

Copy Editor

Mike Henry

Indexer

Ken Johnson

Proofreaders

Kathy Ruiz

Leslie Joseph

Technical Editor

John Lambert

Publishing

Coordinator

Cindy Teeters

Cover and Interior

Designer

Gary Adair

Composition

Mark Shirar

Contents at a Glance

Introduction	1
Part I Introducing the Windows Communication Foundation	
1 Prerequisites	9
2 The Fundamentals	21
3 Data Representation and Durable Services.....	85
4 Sessions, Reliable Sessions, Queues, and Transactions.....	125
Part II Introducing the Windows Workflow Foundation	
5 Fundamentals of the Windows Workflow Foundation.....	147
6 Using the Windows Communication Foundation and the Windows Workflow Foundation Together	209
Part III Security	
7 Security Basics	251
8 Windows CardSpace, Information Cards, and the Identity Metasystem ..	285
9 Securing Applications with Information Cards	329
10 Advanced Security.....	371
Part IV Integration and Interoperability	
11 Legacy Integration	417
12 Interoperability	445
Part V Extending the Windows Communication Foundation	
13 Custom Behaviors	451
14 Custom Channels	485
15 Custom Transports.....	513
Part VI Special Cases	
16 Publish/Subscribe Systems	537
17 Peer Communication	567
18 Representational State Transfer and Plain XML Services	599
Part VII The Lifecycle of Windows Communication Foundation Applications	
19 Manageability.....	623
20 Versioning	661
Part VIII Guidance	
21 Guidance	677
Index	711

Table of Contents

Introduction	1
Part I Introducing the Windows Communication Foundation	
1 Prerequisites	9
Partial Types	9
Generics.....	10
Nullable Value Types.....	13
The Lightweight Transaction Manager.....	14
Role Providers	16
Summary	18
References.....	19
2 The Fundamentals	21
Background	21
Enter Services	24
Windows Communication Foundation	26
The Service Model.....	28
A Software Resource.....	34
Building a Service for Accessing the Resource	36
Using the Service	55
Hosting the Service in IIS	67
Changing How the Service Communicates	72
Visual Studio 2008 Tool Support	75
Summary	82
References.....	83
3 Data Representation and Durable Services	85
Background	85
The XmlSerializer and the DataContractSerializer.....	87
The XML Fetish.....	91
Building a Service	92
Building a Client.....	95
Succumbing to the Urge to Look at XML.....	95
The Case for the DataContractSerializer	95
Using the DataContractSerializer	96
Exception Handling.....	110

Durable Services	114
Why Durable Services?	114
Implementing Durable Services	115
Summary	122
References.....	123
4 Sessions, Reliable Sessions, Queues, and Transactions	125
Reliable Sessions.....	125
Reliable Sessions in Action	127
Session Management	129
Queued Delivery	130
Enhancements in Windows Vista	132
Transactions	134
Summary	143
Part II Introducing the Windows Workflow Foundation	
5 Fundamentals of the Windows Workflow Foundation	147
What Is Windows Workflow Foundation?	147
What Windows Workflow Foundation Is Not	148
Activities.....	149
Out of the Box Activities	151
Creating Custom Activities	152
Communicating with Activities	160
Design Behavior.....	167
Transactions and Compensation.....	170
Workflow Models	172
Sequential Workflows	175
State Machine Workflows.....	183
Custom Root Activities.....	184
Workflow Hosting	184
Hosting the Runtime	185
Runtime Services.....	186
Custom Services.....	196
Rules Engine.....	199
Rules as Conditions	200
The ConditionedActivityGroup Activity.....	202
Rules as Policy.....	204
Summary	207
References.....	207

6	Using the Windows Communication Foundation and the Windows Workflow Foundation Together	209
	Consuming Services.....	210
	Calling Services in a Custom Activity.....	210
	Using the Send Activity (the 3.5 Approach).....	214
	Extending the Send Activity.....	217
	Orchestrating Services.....	219
	Exposing Workflows as Services	220
	Hosting Inside a WCF Service (.NET 3.0).....	220
	Exposing a Workflow as a Service (.NET 3.5).....	226
	Creating a Workflow Service	233
	Context	234
	Patterns of Communication.....	237
	Summary	248
	References.....	248
Part III	Security	
7	Security Basics	251
	Basic Tasks in Securing Communications	251
	Transport Security and Message Security	252
	Using Transport Security.....	253
	Installing Certificates.....	253
	Identifying the Certificate the Server Is to Provide	255
	Configuring the Identity of the Server	256
	Transport Security in Action	257
	Using Message Security.....	263
	Impersonation and Authorization.....	269
	Impersonation.....	269
	Authorization	272
	Reversing the Changes to Windows.....	281
	Uninstalling the Certificates	281
	Removing the SSL Configuration from IIS	282
	Removing the SSL Configuration from HTTP.SYS	283
	Restoring the Identity of the Server	283
	Summary	283
	References.....	284
8	Windows CardSpace, Information Cards, and the Identity Metasystem	285
	The Role of Identity	285
	Microsoft Passport and Other Identity Solutions	288
	The Laws of Identity	290

The Identity Metasystem	291
Information Cards and CardSpace	297
Managing Information Cards	299
Architecture, Protocols, and Security	306
CardSpace and the Enterprise	319
New Features in .NET Framework 3.5	322
HTTP Support in .NET Framework 3.5	324
Summary	326
References.....	327
9 Securing Applications with Information Cards	329
Developing for the Identity Metasystem.....	329
Simple Demonstration of CardSpace.....	331
Prerequisites for the CardSpace Samples	332
1) Enable Internet Information Services and ASP.NET 2.0	333
2) Get X.509 Certificates	333
3) Import the Certificates into the Certificate Store.....	334
4) Update the Hosts File with DNS Entries to Match the Certificates	334
5) Internet Information Services Setup	335
6) Certificate Private Key Access	335
7) HTTP Configuration	336
Adding Information Cards to a WCF Application.....	337
Adding Information Cards	342
Using a Federation Binding	347
Catching Exceptions	348
Processing the Issued Token	350
Using the Metadata Resolver	351
Adding Information Cards to Browser Applications.....	353
Creating a Managed Card	364
Building a Simple Security Token Service	367
Using CardSpace over HTTP	370
Summary	370
References.....	370
10 Advanced Security	371
Prelude.....	371
Securing Resources with Claims	372
Claims-Based Authorization Versus Role-Based Authorization	373
Claims-Based Authorization Versus Access Control Lists	374

Leveraging Claims-Based Security Using XSI	377
Authorizing Access to an Intranet Resource Using Windows Identity	377
Improving the Initial Solution	384
Adding STSs as the Foundation for Federation	391
Reconfiguring the Resource Access Service	405
Reconfiguring the Client	408
Experiencing the Power of Federated, Claims-Based Identity with XSI	411
Claims-Based Security and Federated Security	412
Summary	413
References	414

Part IV Integration and Interoperability

11	Legacy Integration	417
	COM+ Integration.....	417
	Supported Interfaces.....	418
	Selecting the Hosting Mode	419
	Using the COM+ Service Model Configuration Tool.....	419
	Exposing a COM+ Component as a Windows	
	Communication Foundation Web Service.....	421
	Referencing in the Client	426
	Calling a Windows Communication Foundation Service from COM	428
	Building the Service.....	428
	Building the Client	431
	Building the VBScript File	433
	Testing the Solution.....	433
	Integrating with MSMQ.....	433
	Creating a Windows Communication Foundation Service	
	That Integrates with MSMQ.....	434
	Creating the Request	434
	Creating the Service.....	435
	Creating the Client.....	438
	Testing.....	442
	Summary	443
12	Interoperability	445
	Summary	448
	References.....	448

Part V Extending the Windows Communication Foundation

13 Custom Behaviors	451
Extending the Windows Communication Foundation	451
Extending the Service Model with Custom Behaviors	452
Declare What Sort of Behavior You Are Providing	453
Attach the Custom Behavior to an Operation or Endpoint	457
Inform the Windows Communication Foundation of the Custom Behavior	457
Implementing a Custom Behavior	458
Declare the Behavior	458
Attach	458
Inform	459
Implementing Each Type of Custom Behavior	467
Operation Selector	467
Parameter Inspector	469
Message Formatter	471
Message Inspector	473
Instance Context Provider	476
Instance Provider	477
Operation Invokers	478
Implementing a WSDL Export Extension	479
Implementation Steps	480
Custom Behaviors in Action	482
Summary	483
References	483
14 Custom Channels	485
Binding Elements	485
Outbound Communication	486
Inbound Communication	487
Channels Have Shapes	488
Channels Might Be Required to Support Sessions	490
Matching Contracts to Channels	490
Communication State Machines	492
Building Custom Binding Elements	493
Understand the Starting Point	493
Provide a Custom Binding Element That Supports Outbound Communication	495
Amend the Custom Binding Element to Support Inbound Communication	502

Applying a Custom Binding Element Through Configuration	508
Summary	511
15 Custom Transports	513
Transport Channels.....	513
Inbound Communication	514
Outbound Communication	514
Message Encoders.....	514
Completing the Stack	514
Implementing a Transport Binding Element and an Encoder Binding Element.....	516
The Scenario	516
The Requirements.....	517
The TcpListener and the TcpClient Classes.....	517
Implementing Custom Binding Elements to Support an Arbitrary TCP Protocol	520
The Configuration	520
The Custom Transport Binding Element	522
The Channel Listener	525
The Transport Channel	528
The Message Encoder.....	530
Using the Custom Transport Binding Element.....	532
Summary	532
References.....	533
Part VI Special Cases	
16 Publish/Subscribe Systems	537
Publish/Subscribe Using Callback Contracts.....	538
Publish/Subscribe Using MSMQ Pragmatic Multicasting	544
Publish/Subscribe Using Streaming	552
The Streamed Transfer Mode.....	553
Transmitting a Custom Stream with the Streamed Transfer Mode.....	557
Implementing Publish/Subscribe Using the Streamed Transfer Mode and a Custom Stream	561
Summary	565
References.....	566

17	Peer Communication	567
	Using Structured Data in Peer-to-Peer Applications	567
	Leveraging the Windows Peer-to-Peer Networking	
	Development Platform	568
	Understanding Windows Peer-to-Peer Networks	569
	Using Peer Channel	569
	Endpoints.....	569
	Binding	570
	Address.....	574
	Contract	574
	Implementation.....	575
	Peer Channel in Action	575
	Envisaging the Solution	575
	Designing the Data Structures.....	579
	Defining the Service Contracts	581
	Implementing the Service Contracts	584
	Configuring the Endpoints	585
	Directing Messages to a Specific Peer.....	587
	Custom Peer Name Resolution.....	590
	Seeing Peer Channel Work	595
	Peer Channel and People Near Me.....	598
	Summary	598
	References.....	598
18	Representational State Transfer and Plain XML Services	599
	Representational State Transfer	599
	REST Services.....	600
	REST Services and Plain XML	600
	The Virtues and Limitations of REST Services.....	601
	Building REST POX Services with the Windows	
	Communication Foundation	602
	The Address of a REST POX Service Endpoint	602
	The Binding of a REST POX Service Endpoint.....	602
	The Contract of a REST POX Service Endpoint	603
	Implementation.....	604
	A Sample Application	604
	RSS and ATOM Syndication in .NET Framework 3.5	609
	JSON	615
	A Sample ASP.NET AJAX+JSON Application	616
	Summary	620
	References.....	620

Part VII The Lifecycle of Windows Communication Foundation Applications

19 Manageability	623
Instrumentation and Tools	624
The Configuration System and the Configuration Editor.....	625
The Service Configuration Editor.....	627
Configurable Auditing of Security Events.....	633
Message Logging, Activity Tracing, and the Service Trace Viewer	636
Performance Counters	647
WMI Provider	649
Completing the Management Facilities	658
Summary	659
20 Versioning	661
Versioning Nomenclature	662
The Universe of Versioning Problems	662
Adding a New Operation.....	662
Changing an Operation.....	664
Deleting an Operation	668
Changing a Binding	669
Deciding to Retire an Endpoint	669
Changing the Address of a Service Endpoint	670
Centralized Lifecycle Management	670
Summary	673
References.....	673

Part VIII Guidance

21 Guidance	677
Adopting the Windows Communication Foundation.....	677
Working with Windows Communication Foundation Addresses	679
Working with Windows Communication Foundation Bindings	681
Working with Windows Communication Foundation Contracts.....	684
Working with Structural Contracts	687
Working with Behavioral Contracts.....	689
Working with Windows Communication Foundation Services.....	691
Ensuring Manageability	695
Working with Windows Communication Foundation Clients	699
Working with Large Amounts of Data	705
Debugging Windows Communication Foundation Applications	707
Summary	708
References.....	709

Acknowledgments

Many people contributed to this book. The authors would like to thank Joe Long, Eric Zinda, Angela Mills, Omri Gazitt, Steve Swartz, Steve Millet, Mike Vernal, Doug Purdy, Eugene Osvetsky, Daniel Roth, Ford McKinstry, Craig McLuckie, Alex Weinert, Shy Cohen, Yasser Shohoud, Kenny Wolf, Anand Rajagopalan, Jim Johnson, Andy Milligan, Steve Maine, Ram Pamulapati, Ravi Rao, Mark Garbara, Andy Harjanto, T. R. Vishwanath, Doug Walter, Martin Gudgin, Marc Goodner, Giovanni Della-Libera, Kirill Gavrylyuk, Krish Srinivasan, Mark Fussell, Richard Turner, Ami Vora, Ari Bixhorn, Steve Cellini, Neil Hutson, Steve DiMarco, Gianpaolo Carraro, Steve Woodward, James Conard, Nigel Watling, Vittorio Bertocci, Blair Shaw, Jeffrey Schlimmer, Matt Tavis, Mauro Ottoviani, John Frederick, Mark Renfrow, Sean Dixon, Matt Purcell, Cheri Clark, Mauricio Ordonez, Neil Rowe, Donovan Follette, Pat Altimore, Tim Walton, Manu Puri, Ed Pinto, Erik Weiss, Suwat Chitphakdibodin, Govind Ramanathan, Ralph Squillace, John Steer, Brad Severtson, Gary Devendorf, Kavita Kamani, George Kremenliev, Somy Srinivasan, Natasha Jethanandani, Ramesh Seshadri, Lorenz Prem, Laurence Melloul, Clemens Vasters, Joal Lowy, John Justice, David Aiken, Larry Buerk, Wenlong Dong, Nicholas Allen, Carlos Figueira, Ram Poornalingam, Mohammed Makarechian, David Cliffe, David Okonak, Atanu Banerjee, Steven Metsker, Antonio Cruz, Steven Livingstone, Vadim Meleshuk, Elliot Waingold, Yann Christensen, Scott Mason, Jan Alexander, Johan Lindfors, Hanu Kommalapati, Steve Johnson, Tomas Restrepo, Tomasz Janczuk, Garrett Serack, Jeff Baxter, Arun Nanda, Luke Melton, and Al Lee.

A particular debt of gratitude is owed to John Lambert for reviewing the drafts. No one is better qualified to screen the text of a book on a programming technology than an experienced professional software tester. Any mistakes in the pages that follow are solely the fault of the writers, however.

The authors are especially grateful for the support of their wives. They are Marta MacNeill, Kathryn Mercuri, Sylvie Watling, and Libby Winkler. Matt, the only parent so far, would also like to thank his daughter, Grace.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: neil.rowe@pearson.com

Mail: Neil Rowe
Executive Editor
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at www.informit.com/title/9780672330247 for convenient access to any updates, downloads, or errata that might be available for this book.

Introduction

The Windows Communication Foundation, which was code-named Indigo, is a technology that allows pieces of software to communicate with one another. There are many other such technologies, including the Component Object Model (COM) and Distributed Component Object Model (DCOM), Remote Method Invocation (RMI), Microsoft Message Queuing (MSMQ), and WebSphere MQ. Each of those works well in a particular scenario, not so well in others, and is of no use at all in some cases. The Windows Communication Foundation is meant to work well in any circumstance in which a Microsoft .NET assembly must exchange data with any other software entity. In fact, the Windows Communication Foundation is meant to always be the very best option. Its performance is at least on par with that of any other alternative and is usually better; it offers at least as many features and probably several more. It is certainly always the easiest solution to program.

Concretely, the Windows Communication Foundation consists of a small number of .NET libraries with several new sets of classes that it adds to the Microsoft .NET Framework class library, for use with version 2.0 and later of the .NET Common Language Runtime. It also adds some facilities for hosting Windows Communication Foundation solutions to the 5.1 and later versions of Internet Information Services (IIS), the web server built into Windows operating systems.

The Windows Communication Foundation is distributed free of charge as part of a set that includes several other technologies, including the Windows Presentation Foundation, which was code-named Avalon, Windows CardSpace, which was code-named InfoCard, and the

Windows Workflow Foundation. Prior to its release, that group of technologies was called WinFX, but it was renamed the .NET Framework 3.0 in June 2006. Despite that name, the .NET Framework 3.0 and 3.5 is still primarily just a collection of classes added to the .NET Framework 2.0 for use with the 2.0 version of the .NET Common Language Runtime, along with some enhancements to the Windows operating system, as shown in Figure I.1.

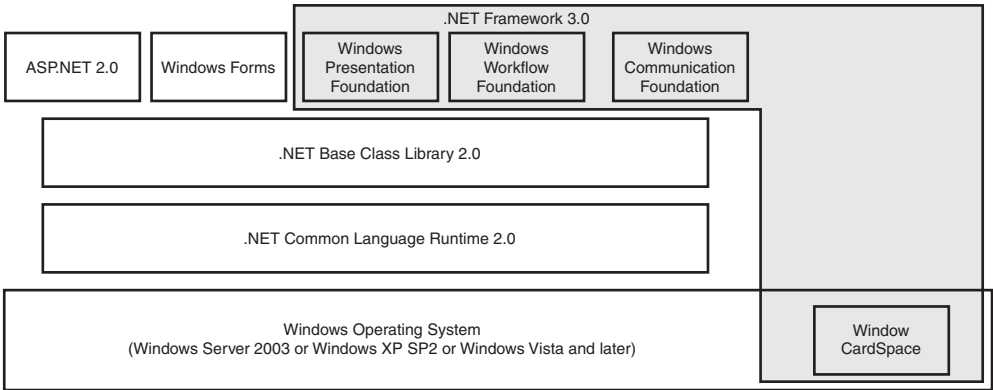


FIGURE I.1 The .NET Framework 3.0.

You can install the .NET Framework 3.0 and 3.5 on Windows XP Service Pack 2, Windows Server 2003, and Windows Server 2003 R2. The runtime components are preinstalled on Windows Vista. On Windows Server 2008 you can add the .NET Framework 3.0 via the Application Server Foundation role service. Only a very small number of features of the .NET Framework 3.0 are available exclusively on Windows Vista and later operating systems.

The .NET Framework 3.5 builds incrementally on top of .NET Framework 3.0. Features relevant to this book include web protocol support for building Windows Communication Foundation services, including AJAX, JSON, REST, POX, RSS and ATOM, workflow-enabled services and full tooling support in Visual Studio 2008. During development, the .NET Framework 3.5 was factored into “red” bits and “green” bits. The red bits were features from .NET Framework 3.0 and the goal was to provide Service Pack levels of compatibility. All the code that worked in 3.0 will work in 3.5. The green bits provide new, additional functionality. Again, the addition of an assembly containing new functionality should have no effect on existing code. The bottom line is that all the code in this book will work with .NET Framework 3.5 and all the code in this book (except the new features introduced in .NET Framework 3.5) should work in .NET Framework 3.0.

This book does not serve as an encyclopedic reference to the Windows Communication Foundation. Instead, it provides the understanding and knowledge required for most practical applications of the technology.

The book explains the Windows Communication Foundation while showing how to use it. So, typically, each chapter provides the precise steps for building a solution that demonstrates a particular aspect of the technology, along with a thorough explanation of each step. Readers who can program in C#, and who like to learn by doing, will be able to follow the steps. Those who prefer to just read will get a detailed account of the features of the Windows Communication Foundation and see how to use them.

To follow the steps in the chapters, you should have installed any version of Visual Studio 2005 or 2008 that includes the C# compiler. Free copies are available at <http://msdn.microsoft.com/vstudio/express/>. You should also have IIS, ASP.NET, and MSMQ installed.

The .NET Framework 3.0 or 3.5 is required, as you might expect. You can download them from <http://www.microsoft.com/downloads/>. The instructions in the chapters assume that all the runtime and developer components of the .NET Framework 3.0 or 3.5 have been installed. It is the runtime components that are preinstalled on Windows Vista and that can be added via the Server Manager on Windows Server 2008. The developer components consist of a Software Development Kit (SDK) and two enhancements to Visual Studio 2005. The SDK provides documentation, some management tools, and a large number of very useful samples. The enhancements to Visual Studio 2005 augment the support provided by IntelliSense for editing configuration files, and provide a visual designer for Windows Workflow Foundation workflows. These features are included in Visual Studio 2008.

To fully utilize Windows CardSpace, which is also covered in this book, you should install Internet Explorer 7. Internet Explorer 7 is also available from <http://www.microsoft.com/downloads>.

Starting points for the solutions built in each of the chapters are available for download from the book's companion page on the publisher's website, as well as from <http://www.cryptmaker.com/WindowsCommunicationFoundationUnleashed>. To ensure that Visual Studio does not complain about the sample code being from a location that is not fully trusted, you can, after due consideration, right-click the downloaded archive, choose Properties from the context menu, and click on the button labeled Unblock, shown in Figure I.2, before extracting the files from the archive.

Note that development on the Vista operating system is supported for Visual Studio 2008 and for Visual Studio 2005 with the Visual Studio 2005 Service Pack 1 Update for Windows Vista. This update is also available from <http://www.microsoft.com/downloads>. Developers working with an earlier version of Visual Studio 2005 on the Vista operating system should anticipate some compatibility issues. To minimize those issues, they can do two things. The first is to disable Vista's User Account Protection feature. The second is to always start Visual Studio 2005 by right-clicking on the executable or the shortcut, selecting Run As from the context menu that appears, and selecting the account of an administrator from the Run As dialog.

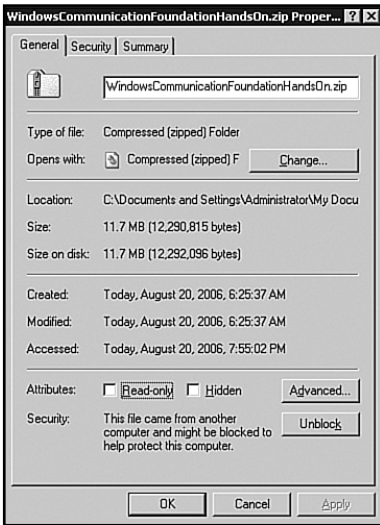


FIGURE I.2 Unlocking a downloaded source code archive.

As with the .NET Framework 3.5 when compared to the .NET Framework 3.0, this book is very similar to its predecessor. Changes include the addition of Visual Studio 2008 support and Chapter 3, “Data Representation and Durable Services,” now covers durable services. The chapters on Windows CardSpace show the updated user interface and cover new features. Chapter 18, “Representational State Transfer and Plain XML Services,” on REST and POX, includes details on the new syndication and JSON APIs. Perhaps the most significant change is a complete rewrite of Chapter 6, “Using the Windows Communication Foundation and the Windows Workflow Foundation Together,” covering the much improved integration between Windows Workflow Foundation and the Windows Communication Foundation.

Many people contributed to this book. The authors would like to thank Joe Long, Eric Zinda, Angela Mills, Omri Gazitt, Steve Swartz, Steve Millet, Mike Vernal, Doug Purdy, Eugene Osvetsky, Daniel Roth, Ford McKinstry, Craig McLuckie, Alex Weinert, Shy Cohen, Yasser Shohoud, Kenny Wolf, Anand Rajagopalan, Jim Johnson, Andy Milligan, Steve Maine, Ram Pamulapati, Ravi Rao, Mark Garbara, Andy Harjanto, T. R. Vishwanath, Doug Walter, Martin Gudgin, Marc Goodner, Giovanni Della-Libera, Kirill Gavrylyuk, Krish Srinivasan, Mark Fussell, Richard Turner, Ami Vora, Ari Bixhorn, Steve Cellini, Neil Hutson, Steve DiMarco, Gianpaolo Carraro, Steve Woodward, James Conard, Nigel Watling, Vittorio Bertocci, Blair Shaw, Jeffrey Schlimmer, Matt Tavis, Mauro Ottoviani, John Frederick, Mark Renfrow, Sean Dixon, Matt Purcell, Cheri Clark, Mauricio Ordonez, Neil Rowe, Donovan Follette, Pat Altimore, Tim Walton, Manu Puri, Ed Pinto, Erik Weiss, Suwat Chitphakdibodin, Govind Ramanathan, Ralph Squillace, John Steer, Brad Severtson, Gary Devendorf, Kavita Kamani, George Kremenliev, Somy Srinivasan, Natasha Jethanandani, Ramesh Seshadri, Lorenz Prem, Laurence Melloul, Clemens Vasters, Joval Lowy, John Justice, David Aiken, Larry Buerk, Wenlong Dong, Nicholas Allen, Carlos

Figueira, Ram Poornalingam, Mohammed Makarechian, David Cliffe, David Okonak, Atanu Banerjee, Steven Metsker, Antonio Cruz, Steven Livingstone, Vadim Meleshuk, Elliot Waingold, Yann Christensen, Scott Mason, Jan Alexander, Johan Lindfors, Hanu Kommalapati, Steve Johnson, Tomas Restrepo, Tomasz Janczuk, Garrett Serack, Jeff Baxter, Arun Nanda, Luke Melton, and Al Lee.

A particular debt of gratitude is owed to John Lambert for reviewing the drafts. No one is better qualified to screen the text of a book on a programming technology than an experienced professional software tester. Any mistakes in the pages that follow are solely the fault of the writers, however.

The authors are especially grateful for the support of their wives. They are Marta MacNeill, Kathryn Mercuri, Sylvie Watling, and Libby Winkler. Matt, the only parent so far, would also like to thank his daughter, Grace.

This page intentionally left blank

PART I

Introducing the Windows Communication Foundation

IN THIS PART

- | | |
|-----------|---|
| CHAPTER 1 | Prerequisites |
| CHAPTER 2 | The Fundamentals |
| CHAPTER 3 | Data Representation and Durable Services |
| CHAPTER 4 | Sessions, Reliable Sessions, Queues, and Transactions |

This page intentionally left blank

CHAPTER 1

Prerequisites

To properly understand and work effectively with the Windows Communication Foundation, you should be familiar with certain facilities of the 2.0 versions of the .NET Framework and the .NET common language runtime. This chapter introduces them: partial types, generics, nullable value types, the Lightweight Transaction Manager, and role providers. The coverage of these features is not intended to be exhaustive, but merely sufficient to clarify their use in the chapters that follow.

Partial Types

Microsoft Visual C# 2005 allows the definition of a type to be composed from multiple partial definitions distributed across any number of source code files for the same module. That option is made available via the modifier `partial`, which can be added to the definition of a class, an interface, or a struct. Therefore, this part of the definition of a class

```
public partial MyClass
{
    private string myField = null;

    public string MyProperty
    {
        get
        {
            return this.myField;
        }
    }
}
```

IN THIS CHAPTER

- ▶ Partial Types
- ▶ Generics
- ▶ Nullable Value Types
- ▶ The Lightweight Transaction Manager
- ▶ Role Providers

and this other part

```
public partial MyClass
{
    public MyClass()
    {
    }

    public void MyMethod()
    {
        this.myField = "Modified by my method.";
    }
}
```

can together constitute the definition of the type `MyClass`. This example illustrates just one use for partial types, which is to organize the behavior of a class and its data into separate source code files.

Generics

“Generics are classes, structures, interfaces, and methods that have placeholders for one or more of the types they store or use” (Microsoft 2006). Here is an example of a generic class introduced in the `System.Collections.Generic` namespace of the .NET Framework 2.0 Class Library:

```
public class List<T>
```

Among the methods of that class is this one:

```
public Add(T item)
```

Here, `T` is the placeholder for the type that an instance of the generic class `System.Collections.Generic.List<T>` will store. In defining an instance of the generic class, you specify the actual type that the instance will store:

```
List<string> myListOfStrings = new List<string>();
```

Then you can use the `Add()` method of the generic class instance like so:

```
myListOfStrings.Add("Hello, World");
```

Generics enable the designer of the `List<T>` class to define a collection of instances of the same unspecified type—in other words, to provide the template for a type-safe collection. A user of `List<T>` can employ it to contain instances of a type of the user’s choosing, without the designer of `List<T>` having to know which type the user might choose. Note as well that whereas a type derived from a base type is meant to derive some of the functionality it requires from the base, with the remainder still having to be programmed, `List<string>` comes fully equipped from `List<T>`.

The class, `System.Collections.Generic.List<T>`, is referred to as a *generic type definition*. The placeholder, `T`, is referred to as a *generic type parameter*. Declaring

```
List<string> myListOfStrings;
```

yields `System.Collections.Generic.List<string>` as a *constructed type*, and `string` as a *generic type argument*.

Generics can have any number of generic type parameters. For example, `System.Collections.Generic.Dictionary<TKey,TValue>` has two.

The designer of a generic may use constraints to restrict the types that can be used as generic type arguments. This generic type definition

```
public class MyGenericType<T> where T: IComparable, new()
```

constrains the generic type arguments to types with a public, parameter-less constructor that implements the `IComparable` interface. This less restrictive generic type definition

```
public class MyGenericType<T> where T: class
```

merely constrains generic type arguments to reference types. Note that `T: class` includes both classes and interfaces.

Both generic and nongeneric types can have generic methods. Here is an example of a nongeneric type with a generic method:

```
using System;
```

```
public class Printer
{
    public void Print<T>(T argument)
    {
        Console.WriteLine(argument.ToString());
    }

    static void Main(string[] arguments)
    {
        Printer printer = new Printer();
        printer.Print<string>("Hello, World");
        Console.WriteLine("Done");
        Console.ReadKey();
    }
}
```

In programming a generic, it is often necessary to determine the type of generic argument that has been substituted for a generic type parameter. This revision to the preceding example shows how you can make that determination:

```

public class Printer
{
    public void Print<T>(T argument)
    {
        if(typeof(T) == typeof(string))
        {
            Console.WriteLine(argument);
        }
        else
        {
            Console.WriteLine(argument.ToString());
        }
    }

    static void Main(string[] arguments)
    {
        Printer printer = new Printer();
        printer.Print<string>("Hello, World");
        Console.WriteLine("Done");
        Console.ReadKey();
    }
}

```

A generic interface may be implemented by a generic type or a nongeneric type. Also, both generic and nongeneric types may inherit from generic base types.

```

public interface IMyGenericInterface<T>
{
    void MyMethod(T input);
}

public class MyGenericImplementation<T>: IMyGenericInterface<T>
{
    public void MyMethod(T input)
    {
    }
}

public class MyGenericDescendant<T> : MyGenericImplementation<T>
{
}

public class MyNonGenericImplementation : IMyGenericInterface<string>
{
    public void MyMethod(string input)
    {
    }
}

```

```
    }  
}  
  
public class MyNonGenericDescendant : MyGenericImplementation<string>  
{  
}  
}
```

Nullable Value Types

According to the Common Language Infrastructure specification, there are two ways of representing data in .NET: by a value type or by a reference type (Ecma 2006, 18). Although instances of value types are usually allocated on a thread's stack, instances of reference types are allocated from the managed heap, and their values are the addresses of the allocated memory (Richter 2002, 134–5).

Whereas the default value of a reference type variable is `null`, indicating that it has yet to be assigned the address of any allocated memory, a value type variable always has a value of the type in question and can never have the value `null`. Therefore, although you can determine whether a reference type has been initialized by checking whether its value is `null`, you cannot do the same for a value type.

However, there are two common circumstances in which you would like to know whether a value has been assigned to an instance of a value type. The first is when the instance represents a value in a database. In such a case, you would like to be able to examine the instance to ascertain whether a value is indeed present in the database. The other circumstance, which is more pertinent to the subject matter of this book, is when the instance represents a data item received from some remote source. Again, you would like to determine from the instance whether a value for that data item was received.

The .NET Framework 2.0 incorporates a generic type definition that provides for cases like these in which you want to assign `null` to an instance of a value type, and test whether the value of the instance is `null`. That generic type definition is `System.Nullable<T>`, which constrains the generic type arguments that may be substituted for `T` to value types. Instances of types constructed from `System.Nullable<T>` can be assigned a value of `null`; indeed, their values are `null` by default. Thus, types constructed from `System.Nullable<T>` are referred to as *nullable value types*.

`System.Nullable<T>` has a property, `Value`, by which the value assigned to an instance of a type constructed from it can be obtained if the value of the instance is not `null`. Therefore, you can write

```
System.Nullable<int> myNullableInteger = null;  
myNullableInteger = 1;  
if (myNullableInteger != null)  
{  
    Console.WriteLine(myNullableInteger.Value);  
}
```

The C# programming language provides an abbreviated syntax for declaring types constructed from `System.Nullable<T>`. That syntax allows you to abbreviate

```
System.Nullable<int> myNullableInteger;
```

to

```
int? myNullableInteger;
```

The compiler will prevent you from attempting to assign the value of a nullable value type to an ordinary value type in this way:

```
int? myNullableInteger = null;
int myInteger = myNullableInteger;
```

It prevents you from doing so because the nullable value type could have the value `null`, which it actually would have in this case, and that value cannot be assigned to an ordinary value type. Although the compiler would permit this code,

```
int? myNullableInteger = null;
int myInteger = myNullableInteger.Value;
```

the second statement would cause an exception to be thrown because any attempt to access the `System.Nullable<T>.Value` property is an invalid operation if the type constructed from `System.Nullable<T>` has not been assigned a valid value of `T`, which has not happened in this case.

One proper way to assign the value of a nullable value type to an ordinary value type is to use the `System.Nullable<T>.HasValue` property to ascertain whether a valid value of `T` has been assigned to the nullable value type:

```
int? myNullableInteger = null;
if (myNullableInteger.HasValue)
{
    int myInteger = myNullableInteger.Value;
}
```

Another option is to use this syntax:

```
int? myNullableInteger = null;
int myInteger = myNullableInteger ?? -1;
```

by which the ordinary integer `myInteger` is assigned the value of the nullable integer `myNullableInteger` if the latter has been assigned a valid integer value; otherwise, `myInteger` is assigned the value of `-1`.

The Lightweight Transaction Manager

In computing, a transaction is a *discrete* activity—an activity that is completed in its entirety or not at all. A resource manager ensures that if a transaction is initiated on some

resource, the resource is restored to its original state if the transaction is not fully completed. A distributed transaction is one that spans multiple resources and therefore involves more than a single resource manager. A manager for distributed transactions has been incorporated into Windows operating systems for many years. It is the *Microsoft Distributed Transaction Coordinator*.

.NET Framework versions 1.0 and 1.1 provided two ways of programming transactions. One way was provided by ADO.NET. That technology's abstract `System.Data.Common.DbConnection` class defined a `BeginTransaction()` method by which you could explicitly initiate a transaction controlled by the particular resource manager made accessible by the concrete implementation of `DbConnection`. The other way of programming a transaction was provided by Enterprise Services. It provided the `System.EnterpriseServices.Transaction` attribute that could be added to any subclass of `System.EnterpriseServices.ServicedComponent` to implicitly enlist any code executing in any of the class's methods into a transaction managed by the Microsoft Distributed Transaction Coordinator.

ADO.NET provided a way of programming transactions explicitly, whereas Enterprise Services allowed you to do it declaratively. However, in choosing between the explicit style of programming transactions offered by ADO.NET and the declarative style offered by Enterprise Services, you were also forced to choose how a transaction would be handled. With ADO.NET, transactions were handled by a single resource manager, whereas with Enterprise Services, a transaction incurred the overhead of involving the Microsoft Distributed Transaction Coordinator, regardless of whether the transaction was actually distributed.

.NET 2.0 introduced the Lightweight Transaction Manager, `System.Transactions.TransactionManager`. As its name implies, the Lightweight Transaction Manager has minimal overhead: "...[p]erformance benchmarking done by Microsoft with SQL Server 2005, comparing the use of a [Lightweight Transaction Manager transaction] to using a native transaction directly found no statistical differences between using the two methods" (Lowy 2005, 12). If only a single resource manager is enlisted in the transaction, the Lightweight Transaction Manager allows that resource manager to manage the transaction and the Lightweight Transaction Manager merely monitors it. However, if the Lightweight Transaction Manager detects that a second resource manager has become involved in the transaction, the Lightweight Transaction Manager has the original resource manager relinquish control of the transaction and transfers that control to the Distributed Transaction Coordinator. Transferring control of a transaction in progress to the Distributed Transaction Coordinator is called *promotion of the transaction*.

The `System.Transactions` namespace allows you to program transactions using the Lightweight Transaction Manager either explicitly or implicitly. The explicit style uses the `System.Transactions.CommitableTransaction` class:

```
CommitableTransaction transaction = new CommitableTransaction();  
using(SqlConnection myConnection = new SqlConnection(myConnectionString))
```

```

{
    myConnection.Open();

    myConnection.EnlistTransaction(tx);

    //Do transactional work

    //Commit the transaction:
    transaction.Close();
}

```

The alternative, implicit style of programming, which is preferable because it is more flexible, uses the `System.Transactions.TransactionScope` class:

```

using(TransactionScope scope = new TransactionScope)
{
    //Do transactional work:
    //...
    //Since no errors have occurred, commit the transaction:
    scope.Complete();
}

```

This style of programming a transaction is implicit because code that executes within the using block of the `System.Transactions.TransactionScope` instance is implicitly enrolled in a transaction. The `Complete()` method of a `System.Transactions.TransactionScope` instance can be called exactly once, and if it is called, the transaction will commit.

The `System.Transactions` namespace also provides a means for programming your own resource managers. However, knowing the purpose of the Lightweight Transaction Manager and the implicit style of transaction programming provided with the `System.Transactions.TransactionScope` class will suffice for the purpose of learning about the Windows Communication Foundation.

Role Providers

Role providers are classes that derive from the abstract class `System.Web.Security.RoleProvider`. That class has the interface shown in Listing 1.1. It defines ten simple methods for managing roles, including ascertaining whether a given user has been assigned a particular role. Role providers, in implementing those abstract methods, will read and write a particular store of role information. For example, one of the concrete implementations of `System.Web.Security.RoleProvider` included in the .NET Framework 2.0 is `System.Web.Security.AuthorizationStoreRoleProvider`, which uses an Authorization Manager Authorization Store as its repository of role information. Another concrete implementation, `System.Web.Security.SqlRoleProvider`, uses a SQL Server database as its store. However, because the `System.Web.Security.RoleProvider` has

such a simple set of methods for managing roles, if none of the role providers included in the .NET Framework 2.0 is suitable, you can readily provide your own implementation to use whatever store of role information you prefer. Role providers hide the details of how role data is stored behind a simple, standard interface for querying and updating that information. Although `System.Web.Security.RoleProvider` is included in the `System.Web` namespaces of ASP.NET, role providers can be used in any .NET 2.0 application.

LISTING 1.1 `System.Web.Security.RoleProvider`

```
public abstract class RoleProvider : ProviderBase
{
    protected RoleProvider();

    public abstract string ApplicationName { get; set; }

    public abstract void AddUsersToRoles(
        string[] usernames, string[] roleNames);
    public abstract void CreateRole(
        string roleName);
    public abstract bool DeleteRole(
        string roleName, bool throwOnPopulatedRole);
    public abstract string[] FindUsersInRole(
        string roleName, string usernameToMatch);
    public abstract string[] GetAllRoles();
    public abstract string[] GetRolesForUser(
        string username);
    public abstract string[] GetUsersInRole(
        string roleName);
    public abstract bool IsUserInRole(
        string username, string roleName);
    public abstract void RemoveUsersFromRoles(
        string[] usernames, string[] roleNames);
    public abstract bool RoleExists(string roleName);
}
```

The static class, `System.Web.Security.Roles`, provides yet another layer of encapsulation for role management. Consider this code snippet:

```
if (!Roles.IsUserInRole(userName, "Administrator"))
{
    [...]
}
```

Here, the static `System.Web.Security.Roles` class is used to inquire whether a given user has been assigned to the Administrator role. What is interesting about this snippet is that the inquiry is made without an instance of a particular role provider having to be created

first. The static `System.Web.Security.Roles` class hides the interaction with the role provider. The role provider it uses is whichever one is specified as being the default in the configuration of the application. Listing 1.2 is a sample configuration that identifies the role provider named `MyRoleProvider`, which is an instance of the `System.Web.Security.AuthorizationStoreRoleProvider` class, as the default role provider.

LISTING 1.2 Role Provider Configuration

```
<configuration>
  <connectionStrings>
    <add name="AuthorizationServices"
      connectionString="msxml://~\App_Data\SampleStore.xml" />
  </connectionStrings>
  <system.web>
    <roleManager defaultProvider="MyRoleProvider"
      enabled="true"
      cacheRolesInCookie="true"
      cookieName=".ASPROLES"
      cookieTimeout="30"
      cookiePath="/"
      cookieRequireSSL="false"
      cookieSlidingExpiration="true"
      cookieProtection="All" >
      <providers>
        <clear />
        <add
          name="MyRoleProvider"
          type="System.Web.Security.AuthorizationStoreRoleProvider"
          connectionStringName="AuthorizationServices"
          applicationName="SampleApplication"
          cacheRefreshInterval="60"
          scopeName="" />
      </providers>
    </roleManager>
  </system.web>
</configuration>
```

Summary

This chapter introduced some programming tools that were new in .NET 2.0 and that are prerequisites for understanding and working effectively with the Windows Communication Foundation:

- The new `partial` keyword in C# allows the definitions of types to be composed from any number of parts distributed across the source code files of a single module.

- ▶ Generics are templates from which any number of fully preprogrammed classes can be created.
- ▶ Nullable value types are value types that can be assigned a value of `null` and checked for `null` values.
- ▶ The Lightweight Transaction Manager ensures that transactions are managed as efficiently as possible. An elegant new syntax has been provided for using it.
- ▶ Role providers implement a simple, standard interface for managing the roles to which users are assigned that is independent of how the role information is stored.

References

Ecma International. 2006. ECMA-335: Common Language Infrastructure (CLI) Partitions I–VI. Geneva: Ecma.

Lowy, Juval. *Introducing System.Transactions*.

<http://www.microsoft.com/downloads/details.aspx?familyid=11632373-BC4E-4C14-AF25-0F32AE3C73A0&displaylang=en>. Accessed July 27, 2008.

Microsoft 2006. *Overview of Generics in the .NET Framework*. <http://msdn2.microsoft.com/en-us/library/ms172193.aspx>. Accessed August 20, 2006.

Richter, Jeffrey. 2002. *Applied Microsoft .NET Framework Programming*. Redmond, WA: Microsoft Press.

This page intentionally left blank

CHAPTER 2

The Fundamentals

Background

Dealing with something as an integrated whole is self-evidently easier than having to understand and manipulate all of its parts. Thus, to make programming easier, it is commonplace to define classes that serve as integrated wholes, keeping their constituents hidden. Doing so is called *encapsulation*, which is characteristic of what is known as *object-oriented programming*.

The C++ programming language provided syntax for encapsulation that proved very popular. In C++, you can write a class like this one:

```
class Stock
{
private:
    char symbol[30];
    int number;
    double price;
    double value;
    void SetTotal()
    {
        this->value = this->number * this->price;
    }
public:
    Stock(void);
    ~Stock(void);
    void Acquire(const char* symbol, int number,
double price);
    void Sell(int number, double price);
};
```

IN THIS CHAPTER

- Background
- Enter Services
- Windows Communication Foundation
- The Service Model
- Visual Studio 2008 Tool Support

The class hides away its data members—symbol, number, price, and value—as well as the method `SetTotal()`, but exposes the methods `Acquire()` and `Sell()` for use.

Some refer to the exposed surface of a class as its *interface*, and to invocations of the methods of a class as *messages*. David A. Taylor does so in his book *Object-Oriented Information Systems: Planning and Integration* (1992, 118).

Using C++ classes to define interfaces and messages has an important shortcoming, however, as Don Box explains in *Essential COM* (1998, 11). There is no standard way for C++ compilers to express the interfaces in binary format. Consequently, sending a message to a class in a dynamic link library (DLL) is not guaranteed to work if the calling code and the intended recipient class were built using different compilers.

That shortcoming is significant because it restricts the extent to which the class in the DLL can be reused in code written by other programmers. The reuse of code written by one programmer within code written by another is fundamental not only to programming productivity, but also to software as a commercial enterprise, to being able to sell what a programmer produces.

Two important solutions to the problem were pursued. One was to define interfaces using C++ abstract base classes. An *abstract base class* is a class with pure virtual functions, and, as Box explains, “[t]he runtime implementation of virtual functions in C++ takes the [same] form...in virtually all production compilers” (1998, 15). You can write, in C++, the code given in Listing 2.1.

LISTING 2.1 Abstract Base Class

```
//IStock.h
class IStock
{
public:
    virtual void DeleteInstance(void);
    virtual void Acquire(const char* symbol, int number, double price) = 0;
    virtual void Sell(int number, double price) = 0;
};

extern "C"
IStock* CreateStock(void);

//Stock.h
#include "IStock.h"

class Stock: public IStock
{
private:
    char symbol[30];
    int number;
    double price;
```

```
double value;
void SetTotal()
{
    this->value = this->number * this->price;
}
public:
    Stock(void);
    ~Stock(void);
    void DeleteInstance(void);
    void Acquire(const char* symbol, int number, double price);
    void Sell(int number, double price);
};
```

In that code, `IStock` is an interface defined using a C++ abstract virtual class. `IStock` is an abstract virtual class because it has the pure virtual functions `Acquire()` and `Sell()`, their nature as pure virtual functions being denoted by having both the keyword, `virtual`, and the suffix, `= 0`, in their declarations. A programmer wanting to use a class with the `IStock` interface within a DLL can write code that retrieves an instance of such a class from the DLL using the global function `CreateStock()` and sends messages to that instance. That code will work even if the programmer is using a different compiler than the one used by the programmer of the DLL.

Programming with interfaces defined as C++ abstract virtual classes is the foundation of a Microsoft technology called the *Component Object Model*, or COM. More generally, it is the foundation of what became known as *component-oriented programming*.

Component-oriented programming is a style of software reuse in which the interface of the reusable class consists of constructors, property getter and setter methods, methods, and events. Programmers using the class “...follow[] a pattern of instantiating a type with a default or relatively simple constructor, setting some instance properties, and finally, [either] calling simple instance methods” or handling the instance’s events (Cwalina and Abrams 2006, 237).

Another important solution to the problem of there being no standard way for C++ compilers to express the interfaces of classes in binary format is to define a standard for the output of compilers. The Java Virtual Machine Specification defines a standard format for the output of compilers, called the *class file format* (Lindholm and Yellin 1997, 61). Files in that format can be translated into the instructions specific to a particular computer processor by a Java Virtual Machine. One programmer can provide a class in the class file format to another programmer who will be able to instantiate that class and send messages to it using any compiler and Java Virtual Machine compliant with the Java Virtual Machine Specification.

Similarly, the Common Language Infrastructure Specification defines the Common Intermediate Language as a standard format for the output of compilers (ECMA

International 2005). Files in that format can be translated into instructions to a particular computer processor by Microsoft's Common Language Runtime, which is the core of Microsoft's .NET technology, as well as by Mono.

Enter Services

Despite these ways of making classes written by one programmer reusable by others, the business of software was still restricted. The use of COM and .NET is widespread, as is the use of Java, and software developed using Java cannot be used together easily with software developed using COM or .NET. More importantly, the component-oriented style of software reuse that became prevalent after the introduction of COM, and which was also widely used by Java and .NET programmers, is grossly inefficient when the instance of the class that is being reused is remote, perhaps on another machine, or even just in a different process. It is so inefficient because, in that scenario, each operation of instantiating the object, of assigning values to its properties, and of calling its methods or handling its events, requires communication back and forth across process boundaries, and possibly also over the network.

"Since [the two separate processes] each have their own memory space, they have to copy the data [transmitted between them] from one memory space to the other. The data is usually transmitted as a byte stream, the most basic form of data. This means that the first process must marshal the data into byte form, and then copy it from the first process to the second one; the second process must unmarshal the data back into its original form, such that the second process then has a copy of the original data in the first process." (Hohpe and Woolf 2004, 66).

Besides the extra work involved in marshalling data across the process boundaries,

"...security may need to be checked, packets may need to be routed through switches. If the two processes are running on machines on opposite sides of the globe, the speed of light may be a factor. The brutal truth is that that any inter-process call is orders of magnitude more expensive than an in-process call—even if both processes are on the same machine. Such a performance effect cannot be ignored." (Fowler 2003, 388).

The Web Services Description Language (WSDL) provided a general solution to the first restriction, the problem of using software developed using Java together with software developed using COM or .NET. WSDL provides a way of defining software interfaces using the Extensible Markup Language (XML), a format that is exceptionally widely adopted, and for which processors are readily available. Classes that implement WSDL interfaces are generally referred to as *services*.

Concomitantly, an alternative to component-oriented programming that is suitable for the reuse of remote instances of classes became progressively more common in practice, although writers seem to have had a remarkably difficult time formalizing its tenets. Thomas Erl, for instance, published two vast books ostensibly on the subject, but never managed to provide a noncircuitous definition of the approach in either of them (Erl

2004, 2005). That alternative to component-oriented programming is service-oriented programming.

Service-oriented programming is a style of software reuse in which the reusable classes are services—classes that implement network facing programming interfaces (WSDL being one way to describe these interfaces)—and the services are designed so as to minimize the number of calls to be made to them, by packaging the data to be transmitted back and forth into messages. A message is a particular kind of data transfer object. A *data transfer object* is

“...little more than a bunch of fields and the getters and setters for them. The value of [this type of object] is that it allows you to move several pieces of information over a network in a single call—a trick that’s essential for distributed systems. [...] Other than simple getters and setters, the data transfer object is [...] responsible for serializing itself into some format that will go over the wire.” (Fowler 2003, 401–403).

Messages are data transfer objects that consist of two parts: a header that provides information pertinent to the operation of transmitting the data and a body that contains the actual data to be transmitted. Crucially, the objects into which the content of a message is read on the receiving side are never assumed to be of the same type as the objects from which the content of the message was written on the sending side. Hence, the sender and the receiver of a message do not have to share the same types, and are said to be *loosely coupled* by their shared knowledge of the format of the message, rather than *tightly coupled* by their shared knowledge of the same types (Box 2004). By virtue of being loosely coupled, the sender and the receiver of the message can evolve independently of one another.

A definition of the term *service-oriented architecture* is warranted here for the sake of disambiguation. Service-oriented architecture is an approach to organizing the software of an enterprise by providing service facades for all of that software, and publishing the description for those services in a central repository. One example of a repository is a Universal Description Discovery and Integration (UDDI) registry. Having interfaces to all the software of an enterprise expressed in a standard format and catalogued in a central repository is desirable because then, in theory, its availability for reuse can be known. Also, policies defining requirements, capabilities, and sundry other properties of a service can be associated with the service using WSDL or by making suitable entries in the registry. That fact leads enterprise architects to anticipate the prospect of the registry serving as a central point of control through which they could issue decrees about how every software entity in their organization is to function by associating policies with the services that provide facades for all of their other software resources. Furthermore, measurements of the performance of the services can be published in the registry, too, so the registry could also serve as a monitoring locus.

Note that service-oriented architecture does not refer to the process of designing software that is to be composed from parts developed using service-oriented programming. There are at least two reasons why not. The first is simply because that is not how the term is actually used, and Ludwig Wittgenstein established in his *Philosophical Investigations* that the meaning of terms is indeed determined by how they are customarily used by a

community (Wittgenstein 1958, 93). The second reason is that the community really could not use the term to refer to a process of designing software composed from parts developed using service-oriented programming because the process of doing that is in no way distinct from the process of designing software composed using parts developed in some other fashion. More precisely, the correct patterns to choose as guides in designing the solution would be the same regardless of whether or not the parts of the solution were developed using service-oriented programming.

Now, Microsoft provided support for service-oriented programming to COM programmers with the Microsoft SOAP Toolkit, and to .NET programmers with the classes in the `System.Web.Services` namespace of the .NET Framework Class Library. Additions to the latter were provided by the Web Services Enhancements for Microsoft .NET. Java programmers can use Apache Axis for service-oriented programming.

Windows Communication Foundation

Yet service-oriented programming has been limited by the lack of standard ways of securing message transmissions, handling failures, and coordinating transactions. Standards have now been developed, and the Windows Communication Foundation provides implementations thereof.

So, the Windows Communication Foundation delivers a more complete infrastructure for service-oriented programming than was available to .NET software developers. Providing that infrastructure is important because service-oriented programming transcends limits on the reuse of software between Java programmers and COM and .NET programmers that had been hampering the software business.

Today, even with the Windows Communication Foundation, service-oriented programming is still suitable only for interactions with remote objects, just as component-oriented programming is suitable only for interacting with local objects. A future goal for the team that developed the Windows Communication Foundation is to extend the technology to allow the service-oriented style of programming to be equally efficient for both scenarios.

Even so, to understand the Windows Communication Foundation as merely being Microsoft's once and future infrastructure for service-oriented programming severely underestimates its significance. The Windows Communication Foundation provides something far more useful than just another way of doing service-oriented programming. It provides a software factory template for software communication.

The concept of software factory templates is introduced by Jack Greenfield and Keith Short in their book *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools* (Greenfield and others 2004). It provides a new approach to model-driven software development.

The notion of model-driven software development has been popular for many years. It is the vision of being able to construct a model of a software solution from which the software itself can be generated after the model has been scrutinized to ensure that it covers

all the functional and nonfunctional requirements. That vision has been pursued using general-purpose modeling languages, the Unified Modeling Language (UML) in particular.

A serious shortcoming in using general-purpose modeling languages for model-driven software development is that general-purpose modeling languages are inherently imprecise. They cannot represent the fine details of requirements that can be expressed in a natural language such as English. They also are not sufficiently precise to cover things such as memory management, thread synchronization, auditing, and exception management. If they were, they would be programming languages, rather than general-purpose modeling languages, yet memory management, thread synchronization, auditing, and exception management are precisely the sorts of things that bedevil programmers.

Greenfield and Short argue that progress in model-driven development depends on eschewing general-purpose modeling languages in favor of *domain-specific languages*, or DSLs. A DSL models the concepts found in a specific domain. DSLs should be used in conjunction with a corresponding class framework, a set of classes specifically designed to cover the same domain. Then, if the DSL is used to model particular ways in which those classes can be used, it should be possible to generate the software described in the model from the class framework (Greenfield and others 2004, 144).

The combination of a DSL and a corresponding class framework constitute the core of a software factory template (Greenfield and others 2004, 173). Software factory templates serve as the software production assets of a software factory from which many varieties of the same software product can be readily fabricated.

A fine example of a software factory template is the Windows Forms Designer in Microsoft Visual Studio .NET and subsequent versions of Microsoft Visual Studio. In that particular case, the Windows Forms Designer is the DSL, the Toolbox and Property Editor being among the terms of the language, and the classes in the `System.Windows.Forms` namespace of the .NET Framework Class Library constitute the class framework. Users of the Windows Forms Designer use it to model software that is generated from those classes.

Programmers have been using the Windows Forms Designer and tools like it in other integrated development environments for many years to develop software user interfaces. So, Greenfield and Short, in introducing the concept of software factory templates, are not proposing a new approach. Rather, they are formalizing one that has already proven to be very successful, and suggesting that it be used to develop other varieties of software besides user interfaces.

The Windows Communication Foundation is a software factory template for software communication. It consists of a DSL, called the *Service Model*, and a class framework, called the *Channel Layer*. The Service Model consists of the classes of the `System.ServiceModel` namespace, and an XML configuration language. The Channel Layer consists of the classes in the `System.ServiceModel.Channel` namespace. Developers model how a piece of software is to communicate using the Service Model, and the communication components they need to have included in their software are generated from the Channel Layer, in

accordance with their model. Later, if they need to change or supplement how their software communicates, they make alterations to their model, and the modifications or additions to their software are generated. If they want to model a form of communication that is not already supported by the Channel Layer, they can build or buy a suitable channel to add to the Channel Layer, and proceed to generate their software as usual, just as a user of the Windows Forms Designer can build or buy controls to add to the Windows Forms Designer's Toolbox.

That the Windows Communication Foundation provides a complete infrastructure for service-oriented programming is very nice because sometimes programmers do need to do that kind of programming, and service-oriented programming will likely remain popular for a while. However, software developers are always trying to get pieces of software to communicate, and they always will need to do that because software is reused by sending and receiving data, and the business of software depends on software reuse. So, the fact that the Windows Communication Foundation provides a software factory template for generating, modifying, and supplementing software communication facilities from a model is truly significant.

The Service Model

The key terms in the language of the Windows Communication Foundation Service Model correspond closely to the key terms of WSDL. In WSDL, a piece of software that can respond to communications over a network is called a *service*. A service is described in an XML document with three primary sections:

- ▶ The service section indicates where the service is located.
- ▶ The binding section specifies which of various standard communication protocols the service understands.
- ▶ The third primary section, the *portType* section, lists all the operations that the service can perform by defining the messages that it will emit in response to messages it receives.

Thus, the three primary sections of a WSDL document tell you where a service is located, how to communicate with it, and what it will do.

Those three things are exactly what you specify in using the Windows Communication Foundation Service Model to model a service: where it is, how to communicate with it, and what it will do. Instead of calling those things *service*, *binding*, and *portType*, as they are called in the WSDL specification, they are named *address*, *binding*, and *contract* in the Windows Communication Foundation Service Model. Consequently, the handy abbreviation *a, b, c* can serve as a reminder of the key terms of the Windows Communication Foundation Service Model and, thereby, as a reminder of the steps to follow in using it to enable a piece of software to communicate.

More precisely, in the Windows Communication Foundation Service Model, a piece of software that responds to communications is a service. A service has one or more endpoints to which communications can be directed. An endpoint consists of an address,

a binding, and a contract. How a service handles communications internally, behind the external surface defined by its endpoints, is determined by a family of control points called *behaviors*.

This chapter explains, in detail, how to use the Windows Communication Foundation Service Model to enable a piece of software to communicate. Lest the details provided obscure how simple this task is to accomplish, here is an overview of the steps involved.

A programmer begins by defining the contract. That simple task is begun by writing an interface in a .NET programming language:

```
public interface IEcho
{
    string Echo(string input);
}
```

It is completed by adding attributes from the Service Model that designate the interface as a Windows Communication Foundation contract, and one or more of its methods as being included in the contract:

```
[ServiceContract]
public interface IEcho
{
    [OperationContract]
    string Echo(string input);
}
```

The next step is to implement the contract, which is done simply by writing a class that implements the interface:

```
public class Service : IEcho
{
    public string Echo(string input)
    {
        return input;
    }
}
```

A class that implements an interface that is designated as a Windows Communication Foundation contract is called a *service type*. How the Windows Communication Foundation conveys data that has been received from the outside via an endpoint to the service type can be controlled by adding behaviors to the service type definition using the `ServiceBehavior` attribute:

```
[ServiceBehavior(ConcurrencyMode=ConcurrencyMode.Multiple)]
public class Service : IEcho
```

```

{
    public string Echo(string input)
    {
        return input;
    }
}

```

For example, the concurrency mode behavior attribute controls whether the Windows Communication Foundation can convey data to the service type on more than one concurrent thread. This behavior is set via an attribute because it is one that a programmer, as opposed to an administrator, should control. After all, it is the programmer of the service type who would know whether the service type is programmed in such a way as to accommodate concurrent access by multiple threads.

The final step for the programmer is to provide for hosting the service within an application domain. IIS can provide an application domain for hosting the service, and so can any .NET application. Hosting the service within an arbitrary .NET application is easily accomplished using the `ServiceHost` class provided by the Windows Communication Foundation Service Model:

```

using (ServiceHost host = new ServiceHost(typeof(Service))
{
    host.Open();

    Console.WriteLine("The service is ready.");
    Console.ReadKey(true);

    host.Close();
}

```

Now the administrator takes over. The administrator defines an endpoint for the service type by associating an address and a binding with the Windows Communication Foundation contracts that the service type implements. An editing tool, called the Service Configuration Editor, shown in Figure 2.1, that also includes wizards, is provided for that purpose.

Whereas the programmer could use attributes to modify the behaviors that a programmer should control, the administrator, as shown in Figure 2.2, can use the Service Configuration Editor to modify the behaviors that are properly within an administrator's purview. All output from the tool takes the form of a .NET application configuration file.

Now that the address, binding, and contract of an endpoint have been defined, the contract has been implemented, and a host for the service has been provided, the service can be made available for use. The administrator executes the host application. The Windows Communication Foundation examines the address, binding, and contract of the endpoint that have been specified in the language of the Service Model, as well as the

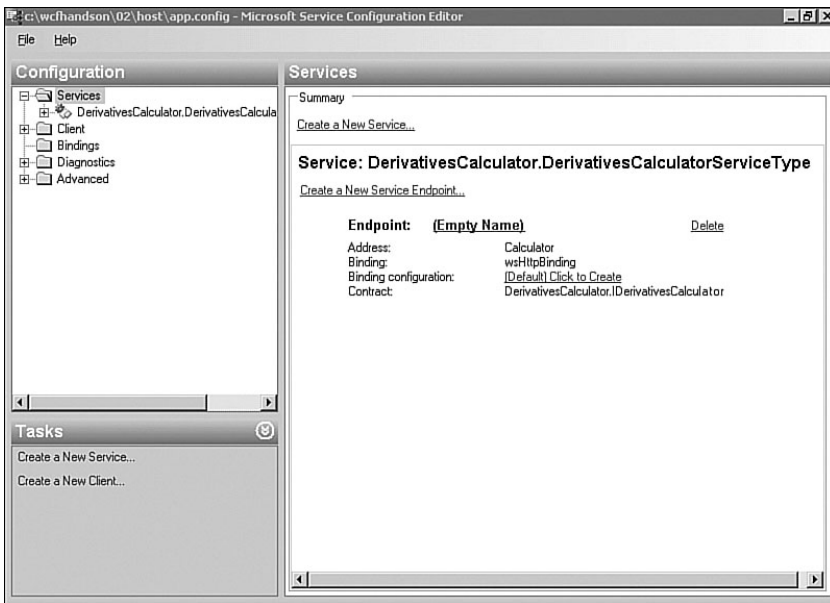


FIGURE 2.1 Defining an address and a binding.

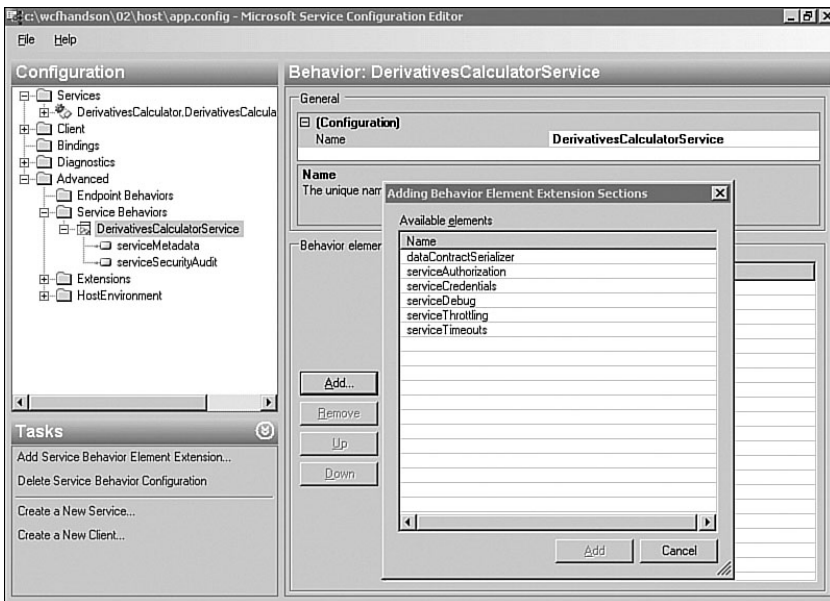


FIGURE 2.2 Adding behaviors to a service.

behaviors, and generates the necessary components by which the service can receive and respond to communications from the Channel Layer.

One of the behaviors that the administrator can control is whether the service exposes WSDL to describe its endpoints for the outside world. If the administrator configures the service to do so, the Windows Communication Foundation will automatically generate the WSDL and offer it up in response to requests for it. A programmer developing an application to communicate with the service can download the WSDL, and generate code for exchanging data with the service, as well as an application configuration file with the endpoint information. That can be done with a single command

```
svcutil http://localhost:8000/EchoService?wsdl
```

wherein `svcutil` is the name of the Windows Communication Foundation's Service Model Metadata Tool, and `http://localhost:8000/EchoService?wsdl` is the address from which the metadata of a service can be downloaded. Having employed the tool to produce the necessary code and configuration, the programmer can proceed to use them to communicate with the service:

```
using(EchoProxy echoProxy = new EchoProxy())
{
    echoProxy.Open();

    string response = echoProxy.Echo("Hello, World!");

    echoProxy.Close();
}
```

The preceding steps are all that is involved in using the Windows Communication Foundation. In summary, those simple steps are merely these:

1. The service programmer defines a contract using a .NET interface.
2. The service programmer implements that interface in a class, a service type.
3. The service programmer optionally modifies Windows Communication Foundation behaviors by applying attributes to the service type or to its methods.
4. The service programmer makes provisions for the service to be hosted. If the service is to be hosted within a .NET application, the programmer develops that application.
5. The service administrator uses the Service Configuration Editor to configure the service's endpoints by associating addresses and bindings with the contracts implemented by the service type.
6. The service administrator optionally uses the Service Configuration Editor to modify Windows Communication Foundation behaviors.
7. The programmer of a client application uses the Service Model Metadata Tool to download WSDL describing the service and to generate code and a configuration file for communicating with the service.

8. The programmer of the client application uses generated code and configuration to exchange data with the service.

At last, the promise of model-driven development might have actually yielded something tangible: a software factory template by which the communications system of an application can be manufactured from a model. The value of using this technology is at least threefold.

First, developers can use the same simple modeling language provided by the Service Model to develop solutions to all kinds of software communications problems. Until now, a .NET developer would typically use .NET web services to exchange data between a .NET application and a Java application, but use .NET Remoting for communication between two .NET applications, and the classes of the `System.Messaging` namespace to transmit data via a queue. The Windows Communication Foundation provides developers with a single, easy-to-use solution that works well for all of those scenarios. Because, as will be shown in Part V, “Extending the Windows Communication Foundation,” the variety of bindings and behaviors supported by the Windows Communication Foundation is indefinitely extensible, the technology will also work as a solution for any other software communication problem that is likely to occur. Consequently, the cost and risk involved in developing solutions to software communications problems decreases because rather than expertise in a different specialized technology being required for each case, the developers’ skill in using the Windows Communication Foundation is reusable in all of them.

Second, in many cases, the administrator is able to modify how the service communicates simply by modifying the binding, without the code having to be changed, and the administrator is thereby able to make the service communicate in a great variety of significantly different ways. The administrator can choose, for instance, to have the same service communicate with clients on the internal network they all share in a manner that is optimal for those clients, and can also have it communicate with Internet clients in a different manner that is suitable for them. When the service’s host executes again after any modifications the administrator has made to the binding, the Windows Communication Foundation generates the communications infrastructure for the new or modified endpoints. Thus, investments in software built using the Windows Communications Foundation yield increased returns by being adaptable to a variety of scenarios.

Third, as will be shown in Part VII, “The Lifecycle of Windows Communication Foundation Applications,” the Windows Communication Foundation provides a variety of powerful tools for managing applications built using the technology. Those tools reduce the cost of operations by saving the cost of having to develop custom management solutions, and by reducing the risk, frequency, duration, and cost of downtime.

The foregoing provides a brief overview of working with the Windows Communication Foundation Service Model. Read on for a much more detailed, step-by-step examination. That account starts right from the beginning, with building some software with which you might like other software to be able to communicate. To be precise, it starts with developing some software to calculate the value of derivatives.

A Software Resource

A *derivative* is a financial entity whose value is derived from that of another. Here is an example. The value of a single share of Microsoft Corporation stock was \$24.41 on October 11, 2005. Given that value, you might offer for sale an option to buy 1,000 of those shares, for \$25 each, one month later, on November 11, 2005. Such an option, which is known as a *call*, might be purchased by someone who anticipates that the price of the shares will rise above \$25 by November 11, 2005, and sold by someone who anticipates that the price of the shares will drop. The call is a derivative, its value being derived from the value of Microsoft Corporation stock.

Pricing a derivative is a complex task. Indeed, estimating the value of derivatives is perhaps the most high-profile problem in modern microeconomics.

In the foregoing example, clearly the quantity of the stock and the current and past prices of the Microsoft Corporation stock are factors to consider. But other factors might be based on analyses of the values of quantities that are thought to affect the prices of the stock, such as the values of various stock market indices, or the interest rate of the U.S. Federal Reserve Bank. In fact, you can say that, in general, the price of a derivative is some function of one or more quantities, one or more market values, and the outcome of one or more quantitative analytical functions.

Although actually writing software to calculate the value of derivatives is beyond the scope of this book, you can pretend to do so by following these steps (note, if you are using VS 2008, one can easily create a service by picking a template under the WCF node in the “Create new project” dialog):

1. Open Microsoft Visual Studio, choose File, New, Project from the menus, and create a new blank solution called `DerivativesCalculatorSolution` in the folder `C:\WCFHandsOn\Fundamentals`, as shown in Figure 2.3.
2. Choose File, New, Project again, and add a C# Class Library project called `DerivativesCalculator` to the solution, as shown in Figure 2.4.
3. Rename the class file `Class1.cs` in the `DerivativesCalculator` project to `Calculator.cs`, and modify its content to look like this:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace DerivativesCalculator
{
    public class Calculator
    {
        public decimal CalculateDerivative(
            string[] symbols,
            decimal[] parameters,
            string[] functions)
        {
```

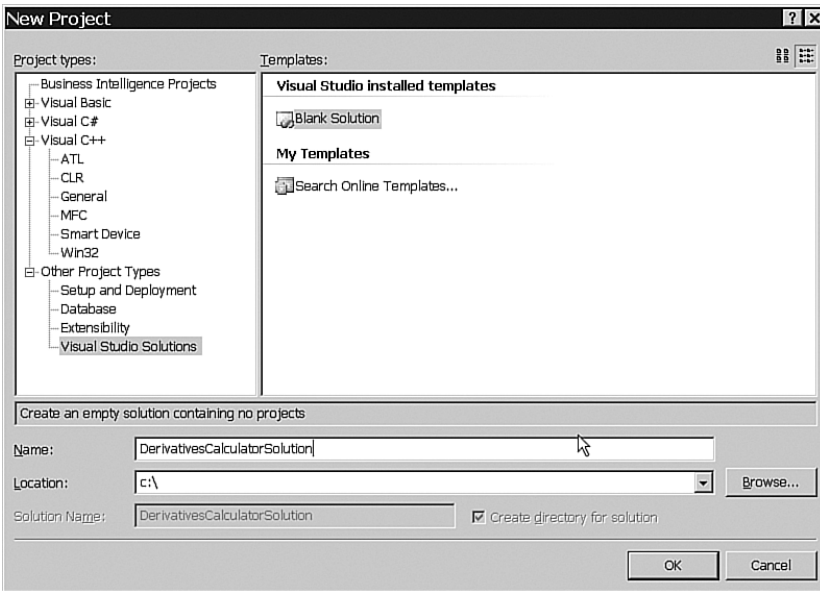


FIGURE 2.3 Creating a blank Visual Studio solution.

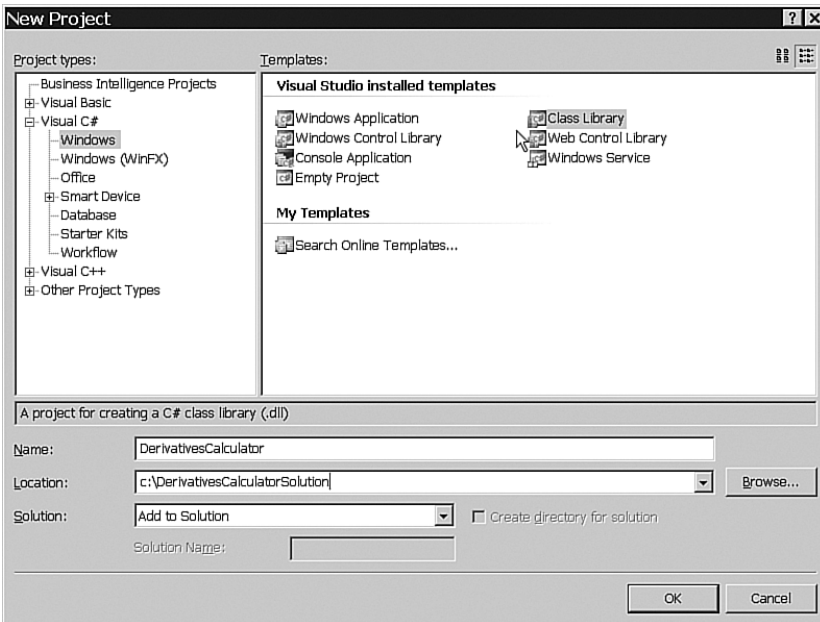


FIGURE 2.4 Adding a Class Library project to the solution.

```

        //Pretend to calculate the value of a derivative.
        return (decimal)(System.DateTime.Now.Millisecond);
    }
}

```

This simple C# class purports to calculate the value of derivatives, and will serve to represent a piece of software with which you might like other software to be able to communicate. Certainly, if the class really could calculate the value of derivatives, its capabilities would be in extraordinary demand, and you could quickly earn a fortune by charging for access to it.

Building a Service for Accessing the Resource

To allow other software to communicate with the class, you can use the Windows Communication Foundation Service Model to add communication facilities to it. You do so by building a Windows Communication Foundation service with an endpoint for accessing the facilities of the derivatives calculator class. Recall that, in the language of the Windows Communication Foundation Service Model, an endpoint consists of an address, a binding, and a contract.

Defining the Contract

In using the Windows Communication Foundation Service Model, you usually begin by defining the contract. The contract specifies the operations that are available at the endpoint. After the contract has been defined, the next step is to implement the contract, to actually provide the operations it defines.

Defining and implementing Windows Communication Foundation contracts is simple. To define a contract, you merely write an interface in your favorite .NET programming language, and adds attributes to it to indicate that the interface is also a Windows Communication Foundation contract. Then, to implement the contract, you simply program a class that implements the .NET interface that you defined. This consists of the following steps:

1. Choose File, New, Project from the Visual Studio menus again, and add another C# Class Library project to the solution, called `DerivativesCalculatorService`.
2. Rename the class file `Class1.cs` in the `DerivativesCalculatorService` project to `IDerivativesCalculator`.
3. Modify the contents of the `IDerivatesCalculator.cs` file to look like so:

```

using System;
using System.Collections.Generic;
using System.Text;

namespace DerivativesCalculator
{
    public interface IDerivativesCalculator
    {

```

```

        decimal CalculateDerivative(
            string[] symbols,
            decimal[] parameters,
            string[] functions);

        void DoNothing();
    }
}

```

IDerivativesCalculator is an ordinary C# interface, with two methods, `CalculateDerivative()` and `DoNothing()`. Now it will be made into a Windows Communication Foundation contract.

4. Choose Project, Add Reference from the Visual Studio menus. Select `System.ServiceModel` from the assemblies listed on the .NET tab of the Add Reference dialog that appears, as shown in Figure 2.5, and click on the OK button. `System.ServiceModel` is the most important of the new .NET class libraries included in the Windows Communication Foundation.

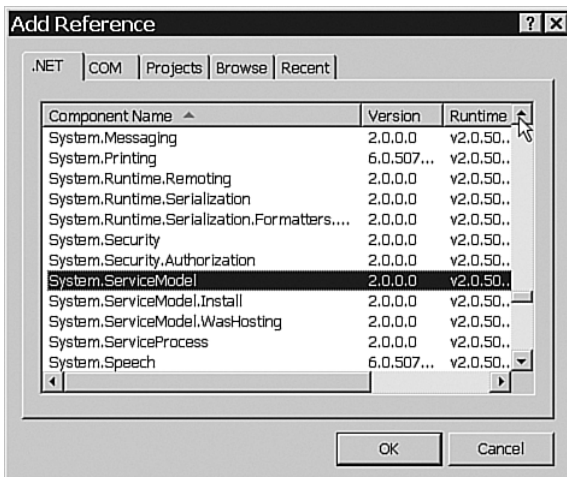


FIGURE 2.5 Adding a reference to the **System.ServiceModel** assembly.

5. Modify the `IDerivativesCalculator` interface in the `IDerivativesCalculator.cs` module to import the classes in the `System.ServiceModel` namespace that is incorporated in the `System.ServiceModel` assembly:

```

using System;
using System.Collections.Generic;
using System.ServiceModel;
using System.Text;

namespace DerivativesCalculator

```

```

{
    public interface IDerivativesCalculator
    {
        decimal CalculateDerivative(
            string[] symbols,
            decimal[] parameters,
            string[] functions);

        void DoNothing();
    }
}

```

6. Now designate the `IDerivativesCalculator` interface as a Windows Communication Foundation contract by adding the `ServiceContract` attribute that is included in the `System.ServiceModel` namespace:

```

using System;
using System.Collections.Generic;
using System.ServiceModel;
using System.Text;

namespace DerivativesCalculator
{
    [ServiceContract]
    public interface IDerivativesCalculator
    {
        decimal CalculateDerivative(
            string[] symbols,
            decimal[] parameters,
            string[] functions);

        void DoNothing();
    }
}

```

7. Use the `OperationContract` attribute to designate the `CalculateDerivative()` method of the `IDerivativesCalculator` interface as one of the methods of the interface that is to be included as an operation in the Windows Communication Foundation contract:

```

using System;
using System.Collections.Generic;
using System.ServiceModel;
using System.Text;

namespace DerivativesCalculator
{
    [ServiceContract]

```

```
public interface IDerivativesCalculator
{
    [OperationContract]
    decimal CalculateDerivative(
        string[] symbols,
        decimal[] parameters,
        string[] functions);

    void DoNothing();
}
}
```

By default, the namespace and name of a Windows Communication Foundation contract are the namespace and name of the interface to which the `ServiceContract` attribute is added. Also, the name of an operation included in a Windows Communication Foundation contract is the name of the method to which the `OperationContract` attribute is added. You can alter the default name of a contract using the `Namespace` and `Name` parameters of the `ServiceContract` attribute, as in

```
[ServiceContract(Namespace="MyNamespace", Name="MyContract")]
public interface IMyInterface
```

You can alter the default name of an operation with the `Name` parameter of the `OperationContract` attribute:

```
[OperationContract(Name="MyOperation")]
string MyMethod();
```

8. Returning to the derivatives calculator solution in Visual Studio, now that a Windows Communication Foundation contract has been defined, the next step is to implement it. In the `DerivativesCalculatorService` project, choose **Project, Add, New Class** from the Visual Studio menus, and add a class called `DerivativesCalculatorServiceType.cs` to the project, as shown in Figure 2.6.
9. Modify the contents of the `DerivativesCalculatorServiceType.cs` class file to look like this:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace DerivativesCalculator
{
    public class DerivativesCalculatorServiceType: IDerivativesCalculator
    {
        #region IDerivativesCalculator Members
        decimal IDerivativesCalculator.CalculateDerivative(
            string[] symbols,
            decimal[] parameters,
```

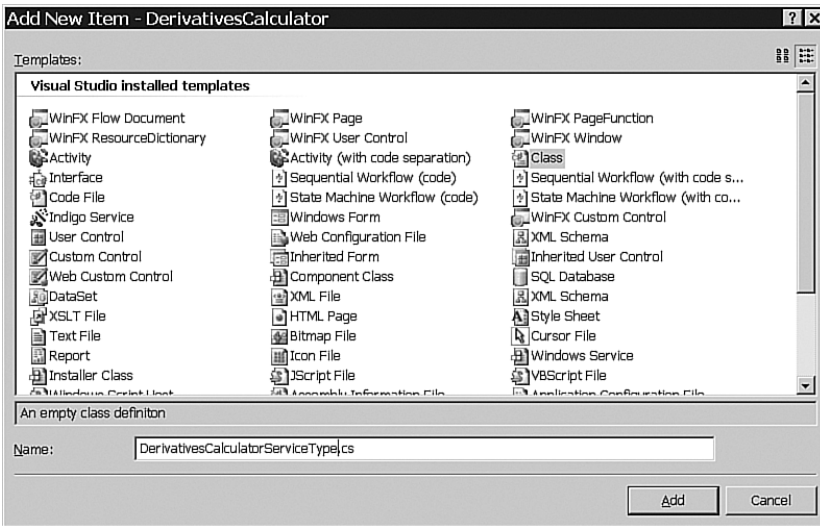


FIGURE 2.6 Adding a class to a project.

```

        string[] functions)
    {
        throw new Exception(
            "The method or operation is not implemented.");
    }

    void IDerivativesCalculator.DoNothing()
    {
        throw new Exception(
            "The method or operation is not implemented.");
    }
    #endregion
}
}

```

As mentioned earlier, in the language of the Windows Communication Foundation, the name *service type* is used to refer to any class that implements a service contract. So, in this case, the `DerivativesCalculatorServiceType` is a service type because it implements the `IDerivativesCalculator` interface, which has been designated as a Windows Communication Foundation service contract.

A class can be a service type not only by implementing an interface that is a service contract, but also by having the `ServiceContract` attribute applied directly to the class. However, by applying the `ServiceContract` attribute to an interface and then implementing the interface with a class, as in the foregoing, you yield a service contract that can be implemented with any number of service types. In particular, one service type that implements the service contract can be discarded in favor of another. If the service contract attribute is instead applied directly to a class, that

class and its descendants will be the only service types that can implement that particular service contract, and discarding the class will mean discarding the service contract.

10. At this point, the `DerivativesCalculatorServiceType` implements the `IDerivativesCalculator` interface in name only. Its methods do not actually perform the operations described in the service contract. Rectify that now by returning to the `DerivativesCalculatorService` project in Visual Studio, and choosing `Project, Add Reference` from the menus. Select the `Projects` tab, select the entry for the `DerivativesCalculator` project, shown in Figure 2.7, and click on the `OK` button.

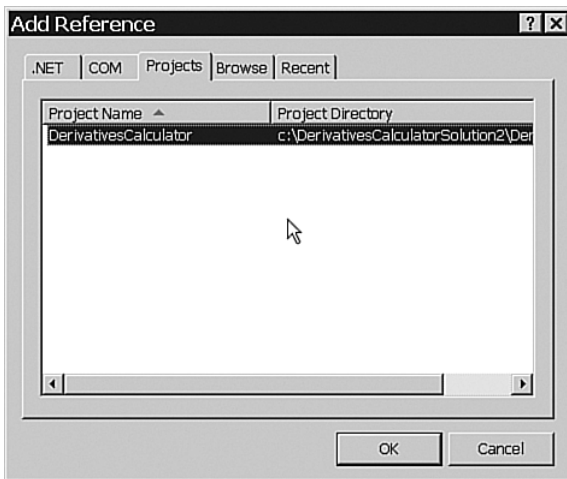


FIGURE 2.7 Adding a reference to the `DerivativesCalculator` project.

11. Now program the `CalculateDerivative()` method of the `DerivativesCalculatorServiceType` to delegate the work of calculating the value of a derivative to the `Calculator` class of the `DerivativesCalculator` project, which was the original class with which other pieces of software were to communicate. Also modify the `DoNothing()` method of the `DerivativesCalculatorServiceType` so that it no longer throws an exception:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace DerivativesCalculator
{
    public class DerivativesCalculatorServiceType: IDerivativesCalculator
    {
        #region IDerivativesCalculator Members
        decimal IDerivativesCalculator.CalculateDerivative(
```

```

        string[] symbols,
        decimal[] parameters,
        string[] functions)
    {
        return new Calculator().CalculateDerivative(
            symbols, parameters, functions);
    }

    void IDerivativesCalculator.DoNothing()
    {
        return;
    }
    #endregion
}
}

```

12. Choose Build, Build Solution from the Visual Studio menu to ensure that there are no programming errors.

Hosting the Service

Recall that the purpose of this exercise has been to use the Windows Communication Foundation to provide a means by which other software can make use of the facilities provided by the derivatives calculator class written at the outset. That requires making a Windows Communication Foundation service by which the capabilities of the derivatives calculator class are made available. Windows Communication Foundation services are collections of endpoints, with each endpoint consisting of an address, a binding, and a contract. At this point, the contract portion of an endpoint for accessing the facilities of the derivatives calculator has been completed, the contract having been defined and implemented.

The next step is to provide for hosting the service within an application domain. Application domains are the containers that Microsoft's Common Language Runtime provides for .NET assemblies. So, in order to get an application domain to host a Windows Communication Foundation service, some Windows process will need to initialize the Common Language Runtime on behalf of the service. Any .NET application can be programmed to do that. IIS can also be made to have Windows Communication Foundation services hosted within application domains. To begin with, the derivatives calculator service will be hosted in an application domain within a .NET application, and then, later, within an application domain in IIS:

1. Choose File, New, Project from the Visual Studio menus, and add a C# console application called Host to the derivatives calculator solution, as shown in Figure 2.8.
2. Select Project, Add Reference from Visual Studio menus, and, from the .NET tab of the Add Reference dialog, add a reference to the `System.ServiceModel` assembly, as

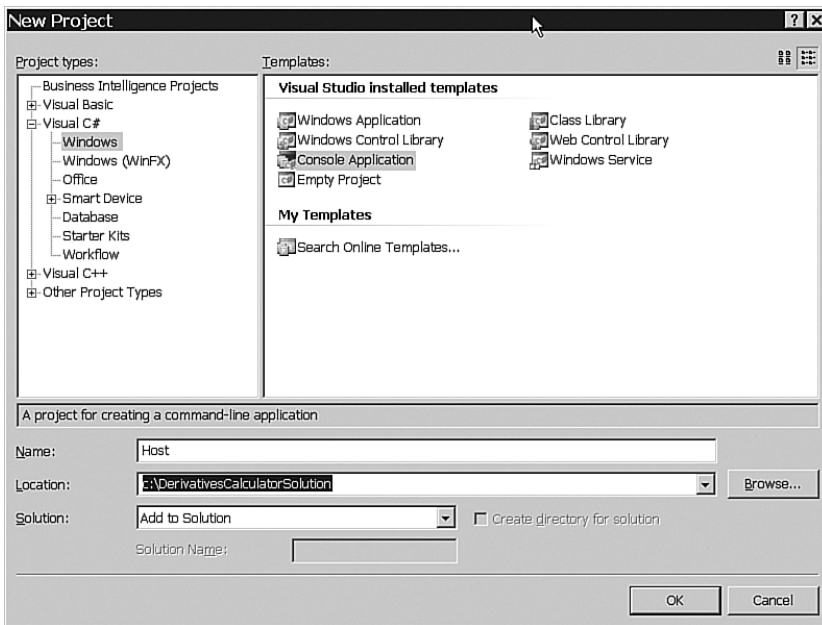


FIGURE 2.8 Adding a Host console application to the solution.

shown earlier in Figure 2.5. Add a reference to the `System.Configuration` assembly in the same way.

3. Choose Project, Add Reference from the Visual Studio menus, and, from the Projects tab, add a reference to the `DerivativesCalculatorService` project.
4. Modify the contents of the `Program.cs` class module in the Host project to match Listing 2.2.

LISTING 2.2 A Host for a Service

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.ServiceModel;
using System.Text;

namespace DerivativesCalculator
{
    public class Program
    {
        public static void Main(string[] args)
        {
```

```

        Type serviceType = typeof(DerivativesCalculatorServiceType);

        using(ServiceHost host = new ServiceHost(
            serviceType))
        {
            host.Open();

            Console.WriteLine(
                "The derivatives calculator service is available."
            );
            Console.ReadKey(true);

            host.Close();
        }
    }
}

```

The key lines in that code are these:

```

using(ServiceHost host = new ServiceHost(
    serviceType))
{
    host.Open();

    ...
    host.Close();
}

```

ServiceHost is the class provided by the Windows Communication Foundation Service Model for programming .NET applications to host Windows Communication Foundation endpoints within application domains. In Listing 2.2, a constructor of the ServiceHost class is given information to identify the service type of the service to be hosted.

5. Choose Build, Build Solution from the Visual Studio menu to ensure that there are no programming errors.

Specifying an Address and a Binding

A Windows Communication Foundation endpoint consists of an address, a binding, and a contract. A contract has been defined and implemented for the endpoint that will be used to provide access to the derivatives calculator class. To complete the endpoint, it is necessary to provide an address and a binding.

Specifying an address and a binding for an endpoint does not require writing any code, and is customarily the work of an administrator rather than a programmer. Providing an address and a binding can be done in code. However, that would require having to modify

the code in order to change the address and the binding of the endpoint. A key innovation of the Windows Communication Foundation is to separate how software is programmed from how it communicates, which is what the binding specifies. So, generally, you avoid the option of specifying the addresses and bindings of endpoints in code, and instead specifies them in configuring the host.

As indicated previously, an editing tool, the Service Configuration Editor, is provided, by which administrators can do the configuration. The use of that tool is covered in detail in Chapter 19, “Manageability.” Here, to facilitate a detailed understanding of the configuration language, the configuration will be done by hand with the following steps:

1. Use the Project, Add Item menu to add an application configuration file named `app.config` to the `DerivativesCalculatorService` project, as shown in Figure 2.9.

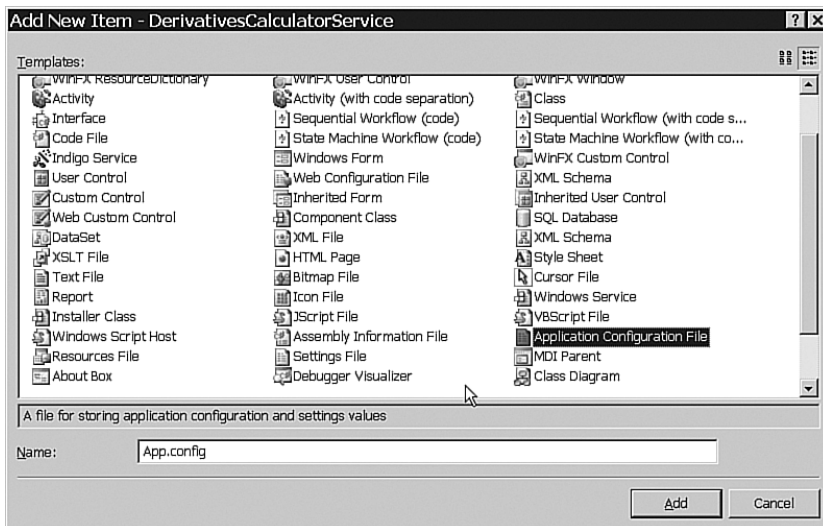


FIGURE 2.9 Adding an application configuration file.

2. Modify the contents of the `app.config` file to look as shown in Listing 2.3.

LISTING 2.3 Adding an Address and a Binding

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name=
"DerivativesCalculator.DerivativesCalculatorServiceType">
        <host>
          <baseAddresses>
            <add baseAddress=
"http://localhost:8000/Derivatives/" />
```

```

        <add baseAddress=
            "net.tcp://localhost:8010/Derivatives/" />
    </baseAddresses>
</host>
<endpoint
    address="Calculator"
    binding="basicHttpBinding"
    contract=
"DerivativesCalculator.IDerivativesCalculator"
    />
</service>
</services>
</system.serviceModel>
</configuration>

```

2. Choose Build, Build Solution from Visual Studio.

In the XML in Listing 2.3,

```

<service name=
"DerivativesCalculator.DerivativesCalculatorServiceType">

```

identifies the service type hosted by the Host application to which the configuration applies. By default, the name by which the service type is identified in the configuration file is matched to the name of a .NET type compiled into the host assembly. In this case, it will be matched to the name of the `DerivativesCalculatorServiceType` class in the `DerivativesCalculator` namespace. Why isn't the service type identified by the name of a type in the standard .NET format, which is called the *assembly-qualified name* format? That format identifies a class not only by its name and namespace, but also by the display name of the assembly containing the type. Those assembly display names consist of the name of the assembly, the version number, a public key token, and a culture identifier. So, the assembly-qualified name of a class might look like this (.NET Framework Class Library 2006):

```

TopNamespace.SubNameSpace.ContainingClass+NestedClass, MyAssembly,
Version=1.3.0.0, Culture=neutral, PublicKeyToken=b17a5c561934e089

```

The assembly-qualified names for types are the standard mechanism for unambiguously identifying a type to be loaded by reflection from any assembly that the Common Language Runtime Loader can locate. Also, they are commonly used in .NET configuration languages for identifying types. Nonetheless, assembly-qualified names are terribly unwieldy to use, for they are not only long, difficult to remember, and easy to mistype, but any errors in entering them also go undetected by the compiler. So, it is a blessing that the Windows Communication Foundation has mostly eschewed their use in its configuration language, and it is to be hoped that the designers of other .NET libraries will

follow that example. Furthermore, although the names used to identify service types in the Windows Communication Foundation configuration language are matched, by default, to the names of types, they are really just strings that custom service hosts can interpret in any fashion, not necessarily treating them as the names of .NET types.

This section of the configuration supplies base addresses for the service host in the form of Uniform Resource Identifiers (URIs):

```
<host>
  <baseAddresses>
    <add baseAddress=
      "http://localhost:8000/Derivatives/" />
    <add baseAddress=
      "net.tcp://localhost:8010/Derivatives/" />
  </baseAddresses>
</host>
```

The addresses provided for the service's endpoints will be addresses relative to these base addresses. The term preceding the initial colon of a URI is called the *scheme*, so the schemes of the two URIs provided as base addresses in this case are `http` and `tcp`. Each base address provided for Windows Communication Foundation services must have a different scheme.

The configuration defines a single endpoint at which the facilities exposed by the service type will be available. The address, the binding, and the contract constituents of the endpoint are all specified:

```
<endpoint
  address="Calculator"
  binding="basicHttpBinding"
  contract="DerivativesCalculator.IDerivativesCalculator"
/>
```

The contract constituent is identified by giving the name of the interface that defines the service contract implemented by the service type. That interface is `IDerivativesCalculator`.

The binding constituent of the endpoint is specified in this way:

```
binding="basicHttpBinding"
```

To understand what that signifies, you must understand Windows Communication Foundation bindings. A Windows Communication Foundation binding defines a combination of protocols for communicating with a service. Each protocol is represented by a single binding element, and a binding is simply a collection of binding elements. Binding

elements are the primary constituents provided by the Windows Communication Foundation's Channel Layer.

One special category of binding element consists of those that implement protocols for transporting messages. One of those is the binding element that implements the Hypertext Transport Protocol (HTTP). Another is the binding element that implements the Transmission Control Protocol (TCP).

Another special category of binding element consists of those that implement protocols for encoding messages. The Windows Communication Foundation provides three such binding elements. One is for encoding SOAP messages as text. Another is for encoding SOAP messages in a binary format. The third is for encoding SOAP messages in accordance with the SOAP Message Transmission Optimization Mechanism (MTOM), which is suitable for messages that incorporate large quantities of binary data.

Examples of Windows Communication Foundation binding elements that are neither transport protocol binding elements nor message-encoding binding elements are the binding elements that implement the WS-Security protocol and the WS-ReliableMessaging protocol. One of the most important ways in which the capabilities of the Windows Communication Foundation can be extended is with the addition of new binding elements that might be provided by Microsoft, or its partners, or by any software developer. Later chapters show how to program custom binding elements, including custom message-encoding binding elements and custom transport protocol binding elements.

A Windows Communication Foundation binding is a set of binding elements that must include at least one transport protocol binding element and zero or more other binding elements. If no message-encoding binding element is specified, the transport protocol binding element will apply its default message-encoding protocol.

Bindings can be defined by selecting individual binding elements, either in code or in configuration. However, the Windows Communication Foundation provides several classes that represent common selections of binding elements. Those classes are referred to as the *predefined bindings*.

One of the predefined bindings is the `BasicHttpBinding`. The `BasicHttpBinding` represents the combination of the HTTP transport binding element and the binding element for encoding SOAP messages in text format. The `BasicHttpBinding` class configures those binding elements in accordance with the WS-I Basic Profile Specification 1.1, which is a combination of web service specifications chosen to promote interoperability among web services and consumers of web services on different platforms.

All the current predefined bindings are listed in Table 2.1. They each derive, directly or indirectly, from the class `System.ServiceModel.Channels.Binding`.

TABLE 2.1 Windows Communication Foundation Predefined Bindings

Name	Purpose
BasicHttpBinding	Maximum interoperability through conformity to the WS-BasicProfile 1.1
WSHttpBinding	HTTP communication in conformity with WS-* protocols
WS2007HttpBinding	HTTP communication in conformity with WS-* protocols, updated to reflect ratified standards
WSDualHttpBinding	Duplex HTTP communication, by which the receiver of an initial message will not reply directly to the initial sender, but may transmit any number of responses over a period
WSFederationBinding	HTTP communication, in which access to the resources of a service can be controlled based on credentials issued by an explicitly identified credential provider
WS2007FederationBinding	HTTP communication, in which access to the resources of a service can be controlled based on credentials issued by an explicitly identified credential provider, updated to reflect ratified standards
NetTcpBinding	Secure, reliable, high-performance communication between Windows Communication Foundation software entities across a network
NetNamedPipeBinding	Secure, reliable, high-performance communication between Windows Communication Foundation software entities on the same machine
NetMsmqBinding	Communication between Windows Communication Foundation software entities via Microsoft Message Queuing (MSMQ)
MsmqIntegrationBinding	Communication between a Windows Communication Foundation software entity and another software entity via MSMQ
NetPeerTcpBinding	Communication between Windows Communication Foundation software entities via Windows Peer-to-Peer Networking
WebHttpBinding	HTTP Communication in conformity with standard HTTP functionality, used with RESTful architectural styles, POX (plain old XML) and JSON (JavaScript Object Notation) services

This specification, in the configuration of the endpoint for the `DerivativesCalculatorService` in Listing 2.3

```
binding="basicHttpBinding"
```

identifies the `BasicHttpBinding` as the binding for that endpoint. The lowercase of the initial letter, `b`, is in conformity with a convention of using camel-casing in configuration files.

You can adjust the settings of a predefined binding by adding a binding configuration to the definition of the endpoint like so:

```
<system.serviceModel>
  <services>
    <service type=
"DerivativesCalculator.DerivativesCalculatorServiceType">
      <endpoint
        address="Calculator"
        binding="basicHttpBinding"
        bindingConfiguration="bindingSettings"
        contract=
"DerivativesCalculator.IDerivativesCalculator"
      />
    </service>
  </services>
  <bindings>
    <basicHttpBinding>
      <binding name="bindingSettings" messageEncoding="Mtom" />
    </basicHttpBinding>
  </bindings>
</system.serviceModel>
```

In this case, the settings for the predefined `BasicHttpBinding` are adjusted so as to use the MTOM message-encoding binding element rather than the default text message-encoding binding element.

The address specified for the endpoint in the configuration of the `DerivativesCalculatorService` in Listing 2.3 is `Calculator`. That address for the endpoint is relative to a base address. Which of the base addresses defined for a service is the base address for the endpoint? It is determined based on the scheme of the base address and

the transport protocol implemented by the transport-binding element of the endpoint, as shown in Table 2.2. The transport protocol implemented by the transport-binding element of the endpoint is the HTTP protocol, so, based on the information in Table 2.2, the base address for the endpoint is `http://localhost:8000/Derivatives/`. Therefore, the absolute address for the endpoint is `http://localhost:8000/Derivatives/Calculator`.

TABLE 2.2 Mapping of Base Address Schemes to Transport Protocols

Base Address Scheme	Transport Protocol
http	HTTP
net.tcp	TCP
net.pipe	Named Pipes
net.msmq	MSMQ

Anyone who would like to know the complete Windows Communication Foundation configuration language should study the XML Schema file containing the definition of the configuration language. Assuming that Visual Studio 2008 has been installed, that XML Schema file should be `\Program Files\Microsoft Visual Studio 9.0\Xml\Schemas\DotNetConfig.xsd`, on the disc where Visual Studio 2008 is installed. If that file seems to be missing, search for a file with the extension `.xsd`, containing the expression `system.serviceModel`.

Deploying the Service

Now an address, a binding, and a contract have been provided for the Windows Communication Foundation endpoint at which the facilities of the derivatives calculator class will be made available. An application domain for hosting the service incorporating that endpoint has also been provided, or, to be more precise, it will be provided as soon as the Host console application is executed:

1. Execute that application now by right-clicking on the Host entry in the Visual Studio Solution Explorer, and selecting Debug, Start New Instance from the context menu. After a few seconds, the console application window of the host should appear, as in Figure 2.10.

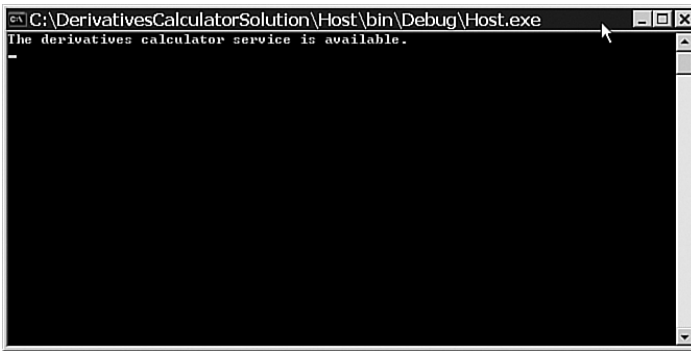


FIGURE 2.10 The Host console application running.

The Windows Communication Foundation has examined the code in the `Host` and `DerivativesCalculatorService` assemblies, as well as the contents of the `Host` assembly's configuration file. The code and the configuration use the Windows Communication Foundation's Service Model to define a service for accessing the derivatives calculator class. From that code and that configuration, the Windows Communication Foundation generates and configures the service using the programming framework constituted by the classes of the Channel Layer. In particular, it employs the binding element classes used by the `BasicProfileBinding` class that was selected as the binding for the service. Then the Windows Communication Foundation loads the service into the default application domain of the Host console application.

This is the step at which folks using the Vista operating system with a version of Visual Studio 2005 that does not have the Vista compatibility update installed could run into difficulty. They might encounter a namespace reservation exception, due to their service being denied the right to use the address they have specified for its endpoint. In that case, it will be necessary for them to grant permission to have a service use the address to the `NETWORK SERVICE` user account. The official tool for that purpose is Microsoft's `Httpcfg.exe`. A more usable one is Steve Johnson's HTTP configuration utility, which, unlike the Microsoft tool and several others for the same job, sports a graphical user interface. His utility is available at <http://www.StevesTechSpot.com>. Using Visual Studio 2008, the autohosting capability will eliminate the need for this step.

2. Confirm that the Windows Communication Foundation service for accessing the capabilities of the derivatives calculator class is available by directing a browser to the HTTP base address that was specified for the service in the host's application configuration file: `http://localhost:8000/Derivatives/`. A page like the one shown in Figure 2.11 should be opened in the browser.

A similar page can be retrieved for any Windows Communications Foundation service with a host that has been provided with a base address with the scheme `http`. It is not necessary that the service have any endpoints with addresses relative to that base address. Note, though, that the page cautions that metadata publishing

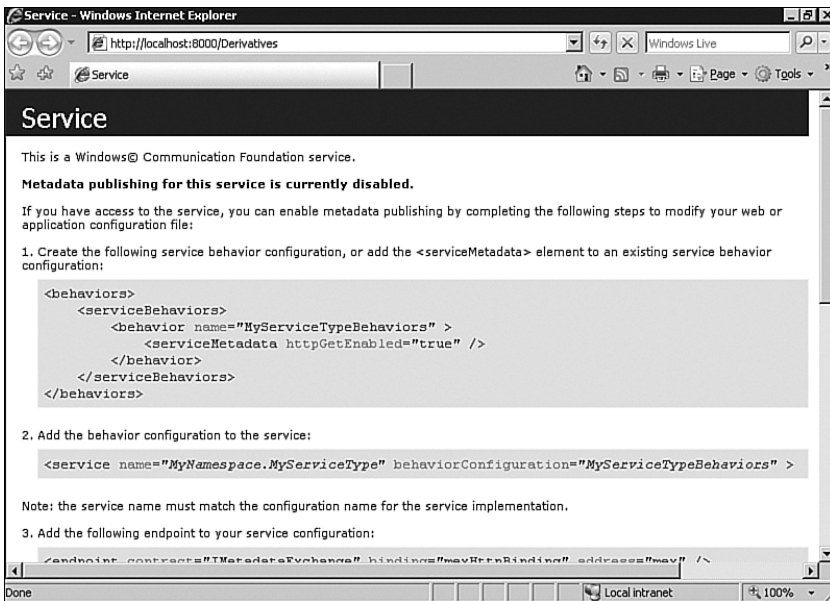


FIGURE 2.11 Help page for a service.

for the service is disabled. Metadata publishing is disabled by default for the sake of maximizing security so that nothing is made known about a service except what is deliberately chosen by its developers and administrators. To deliberately opt to have the service publish its metadata, it will be necessary to reconfigure it.

3. Choose Debug, Stop Debugging from the Visual Studio menus to terminate the instance of the Host console application so that its configuration can be modified.
4. The configuration needs to be modified to include a behavior setting by which metadata publishing is activated. Edit the `app.config` file in the Host project of the DerivativesCalculator Solution so that its content matches that of Listing 2.4.

LISTING 2.4 Enabling Metadata Publication

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name=
"DerivativesCalculator.DerivativesCalculatorServiceType"
        behaviorConfiguration=
          "DerivativesCalculatorService">
        <host>
          <baseAddresses>
            <add baseAddress=
              "http://localhost:8000/Derivatives/" />
            <add baseAddress=
```

```

        "net.tcp://localhost:8010/Derivatives/" />
    </baseAddresses>
</host>
<endpoint
    address="Calculator"
    binding="basicHttpBinding"
    contract=
"DerivativesCalculator.IDerivativesCalculator"
    />
</service>
</services>
<behaviors>
    <serviceBehaviors>
        <behavior name=
            "DerivativesCalculatorService">
            <serviceMetadata
                httpGetEnabled="true" />
        </behavior>
    </serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>

```

The additions to the configuration of the Windows Communication Foundation service signify that behaviors that apply to a service, rather than to one of its endpoints, are being modified:

```

<behaviors>
    <serviceBehaviors>
        ...
    </serviceBehaviors>
</behaviors>

```

The modification is made to the service's `ServiceMetadata` behavior, and the nature of the modification is to have the service generate its metadata in response to a request for it in the form of an HTTP GET:

```

<serviceMetadata httpGetEnabled="true" />

```

To associate this behavior configuration with the `DerivativesCalculator` service, the behavior is given a name

```

<behavior name="DerivativesCalculatorService">
    ...
</behavior>

```

and identified by that name as a behavior configuration that applies to the service:

```

<service name=
"DerivativesCalculator.DerivativesCalculatorServiceType"

```

```
behaviorConfiguration="DerivativesCalculatorService">
    ...
</service>
```

5. Execute the Host console application again by right-clicking on the Host entry in the Visual Studio Solution Explorer, and selecting Debug, Start New Instance from the context menu.
6. Add the query wsdl to the URI at which the browser is pointing, by aiming the browser at <http://localhost:8000/Derivatives/?wsdl>, as in Figure 2.12, and the WSDL for the service should be displayed.

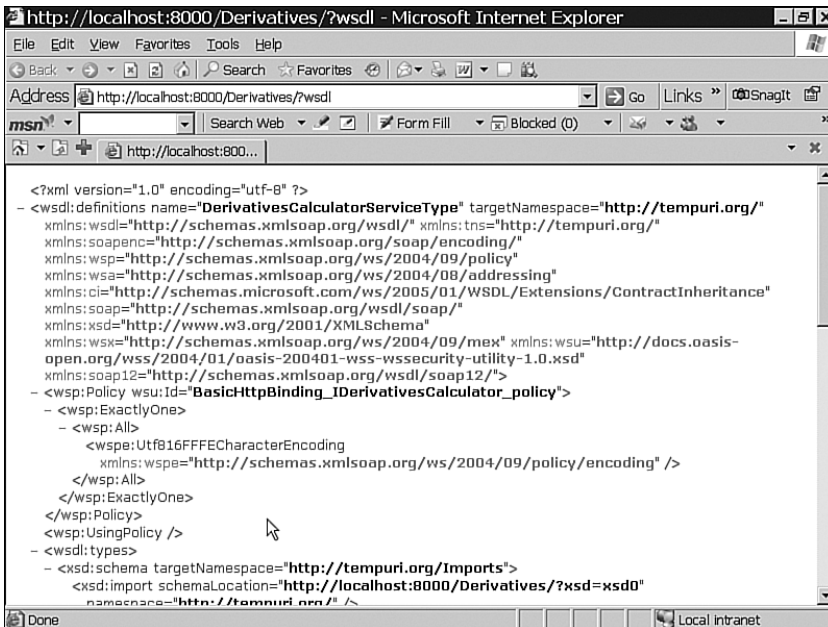


FIGURE 2.12 Examining the WSDL for a service.

Using the Service

Now a Windows Communication Foundation service is available for accessing the facilities of the derivatives calculator class. The Windows Communication Foundation can be employed to construct a client for the derivatives calculator, a software entity that uses the facilities of the derivatives calculator via the service. Different ways to build the client with the Windows Communication Foundation are shown, as well as ways to build a client in Java for the same Windows Communication Foundation service.

Using the Service with a Windows Communication Foundation Client

For the following steps, access to the tools provided with the Windows Communication Foundation from a .NET command prompt will be required. Assuming a complete and

normal installation of the Microsoft Windows SDK for the .NET Framework 3.0, that access is provided by a command prompt that should be accessible from the Windows Start menu by choosing All Programs, Microsoft Windows SDK, CMD Shell. That command prompt will be referred to as the *SDK Command Prompt*. From that prompt, the Windows Communication Foundation's Service Metadata Tool, `SvcUtil.exe`, will be used to generate components of the client for the derivatives calculator as seen in the following steps:

1. If the Host console application had been shut down, start an instance of it, as before.
2. Open the SDK Command Prompt.
3. Enter

```
:C:
```

and then

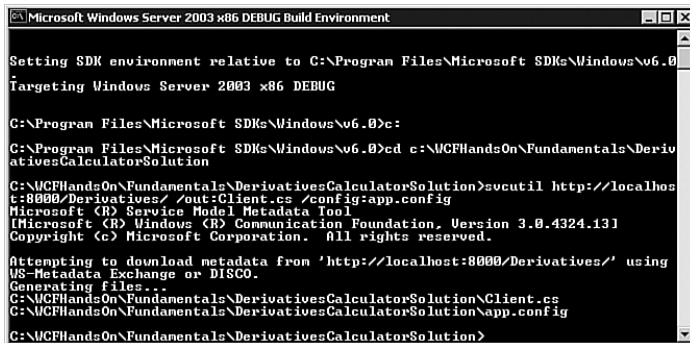
```
:cd c:\WCFHandsOn\Fundamentals\DerivativesCalculatorSolution
```

at that prompt to make the derivatives calculator solution folder the current directory.

4. Next, enter

```
:svcutil http://localhost:8000/Derivatives/ /out:Client.cs /config:app.config
```

The output should be as shown in Figure 2.13.



```

Microsoft Windows Server 2003 x86 DEBUG Build Environment

Setting SDK environment relative to C:\Program Files\Microsoft SDKs\Windows\v6.0
Targeting Windows Server 2003 x86 DEBUG

C:\Program Files\Microsoft SDKs\Windows\v6.0>c:
C:\Program Files\Microsoft SDKs\Windows\v6.0>cd c:\WCFHandsOn\Fundamentals\DerivativesCalculatorSolution
C:\WCFHandsOn\Fundamentals\DerivativesCalculatorSolution>svcutil http://localhost:8000/Derivatives/ /out:Client.cs /config:app.config
Microsoft (R) Service Model Metadata Tool
[Microsoft (R) Windows (R) Communication Foundation, Version 3.0.4324.13]
Copyright (c) Microsoft Corporation. All rights reserved.

Attempting to download metadata from 'http://localhost:8000/Derivatives/' using
WS-Metadata Exchange or DISCO.
Generating files...
C:\WCFHandsOn\Fundamentals\DerivativesCalculatorSolution\Client.cs
C:\WCFHandsOn\Fundamentals\DerivativesCalculatorSolution\app.config
C:\WCFHandsOn\Fundamentals\DerivativesCalculatorSolution>
  
```

FIGURE 2.13 Using the Service Metadata Tool.

The command executes the Windows Communication Foundation's Service Metadata Tool, passing it a base address of a service that has the scheme `http`. In this case, it is passed the base address of the derivatives calculator service constructed by the earlier steps in this chapter. Given a base address of a Windows Communication Foundation service, provided it is an address with the scheme `http`, and provided metadata publishing via HTTP GET is enabled for the service, the Service Metadata Tool can retrieve the WSDL for the service and other associated metadata. By default, it also generates the C# code for a class that can serve as a proxy for communicating with the service, as well as a .NET application-specific configuration file containing

the definition of the service's endpoints. The switches `/out:Client.cs` and `/config:app.config` used in the foregoing command specify the names to be used for the file containing the C# code and for the configuration file. In the next few steps, the output from the Service Metadata Tool will be used to complete the client for the derivatives calculator.

5. Choose Debug, Stop Debugging from the Visual Studio menus to terminate the instance of the Host console application so that the solution can be modified.
6. Select File, New, Project from Visual Studio menus to add a C# Console Application project called Client to the DerivativesCalculator solution.
7. Choose Project, Add Reference from the Visual Studio menus, and add a reference to the Windows Communication Foundation's `System.ServiceModel` .NET assembly to the client project.
8. Select Project, Add Existing Item from the Visual Studio menus, and add the files `Client.cs` and `app.config`, in the folder `C:\WCFHandsOn\Fundamentals\DerivativesCalculatorSolution`, to the Client project, as shown in Figure 2.14. Those are the files that should have been emitted by the Service Metadata Tool.

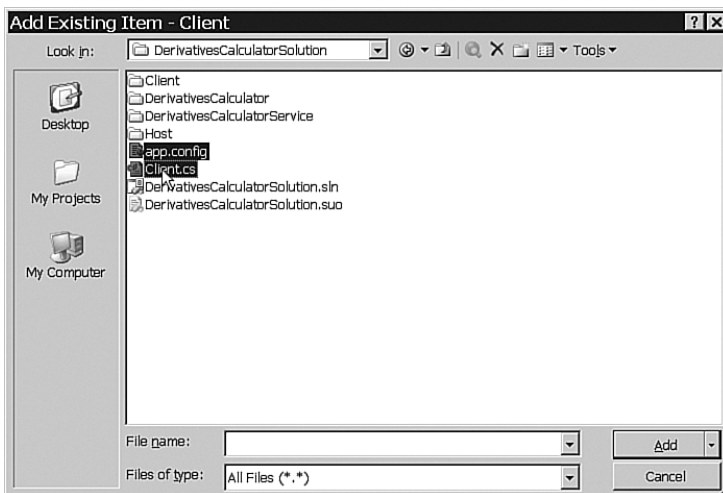


FIGURE 2.14 Adding output from the Service Metadata Tool to a project.

9. Alter the code in the `Program.cs` file of the Client project of the derivatives calculator solution to use the class generated by the Service Metadata Tool as a proxy for communicating with the derivatives calculator service. The code in the `Program.cs` file should be the code in Listing 2.5.

LISTING 2.5 Using the Generated Client Class

```
using System;
```

```

using System.Collections.Generic;
using System.Text;

namespace Client
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Press any key when the service is ready.");
            Console.ReadKey(true);

            decimal result = 0;
            using (DerivativesCalculatorProxy proxy =
                new DerivativesCalculatorProxy(
                    "BasicHttpBinding_IDerivativesCalculator"))
            {
                proxy.Open();
                result = proxy.CalculateDerivative(
                    new string[] { "MSFT" },
                    new decimal[] { 3 },
                    new string[] { });
                proxy.Close();
            }

            Console.WriteLine(string.Format("Result: {0}", result));

            Console.WriteLine("Press any key to exit.");
            Console.ReadKey(true);
        }
    }
}

```

In Listing 2.5, the statement

```

DerivativesCalculatorProxy proxy =
    new DerivativesCalculatorProxy(
        "BasicHttpBinding_IDerivativesCalculator"))

```

creates an instance of the class generated by the Service Metadata Tool to serve as a proxy for the derivatives calculator service. The string parameter passed to the constructor of the class, `BasicHttpBinding_IDerivativesCalculator`, identifies which definition of an endpoint in the application's configuration file is the definition of the endpoint with which this instance of the class is to communicate. Therefore, an endpoint definition in the configuration file must be named accordingly.

The app.config file added to the Client project in step 8 should contain this definition of an endpoint, with a specification of an address, a binding, and a contract:

```
<client>
  <endpoint
    address="http://localhost:8000/Derivatives/Calculator"
    binding="basicHttpBinding"
    bindingConfiguration="BasicHttpBinding_IDerivativesCalculator"
    contract="IDerivativesCalculator"
    name="BasicHttpBinding_IDerivativesCalculator" />
</client>
```

Notice that the name provided for this endpoint definition matches the name that is passed to the constructor of the proxy class.

The binding configuration named `BasicHttpBinding_IDerivativesCalculator` to which this endpoint configuration refers, explicitly specifies the default values for the properties of the predefined `BasicHttpBinding`:

```
<basicHttpBinding>
  <binding
    name="BasicHttpBinding_IDerivativesCalculator"
    closeTimeout="00:01:00"
    openTimeout="00:01:00"
    receiveTimeout="00:10:00"
    sendTimeout="00:01:00"
    allowCookies="false"
    bypassProxyOnLocal="false"
    hostNameComparisonMode="StrongWildcard"
    maxBufferSize="65536"
    maxBufferPoolSize="524288"
    maxReceivedMessageSize="65536"
    messageEncoding="Text"
    textEncoding="utf-8"
    transferMode="Buffered"
    useDefaultWebProxy="true">
    <readerQuotas
      maxDepth="32"
      maxStringContentLength="8192"
      maxArrayLength="16384"
      maxBytesPerRead="4096"
      maxNameTableCharCount="16384" />
  <security mode="None">
    <transport
      clientCredentialType="None"
      proxyCredentialType="None"
      realm="" />
    <message
```

```
clientCredentialType="UserName"
algorithmSuite="Default" />

</security>
</binding>

</basicHttpBinding>
```

Those default values were left implicit in the service’s configuration of the endpoint. The Service Metadata Tool diligently avoided the assumption that the configuration of the binding that it derived from the metadata is the default configuration.

- 10. Prepare to have the client use the derivatives calculator by modifying the startup project properties of the derivatives calculator solution as shown in Figure 2.15.

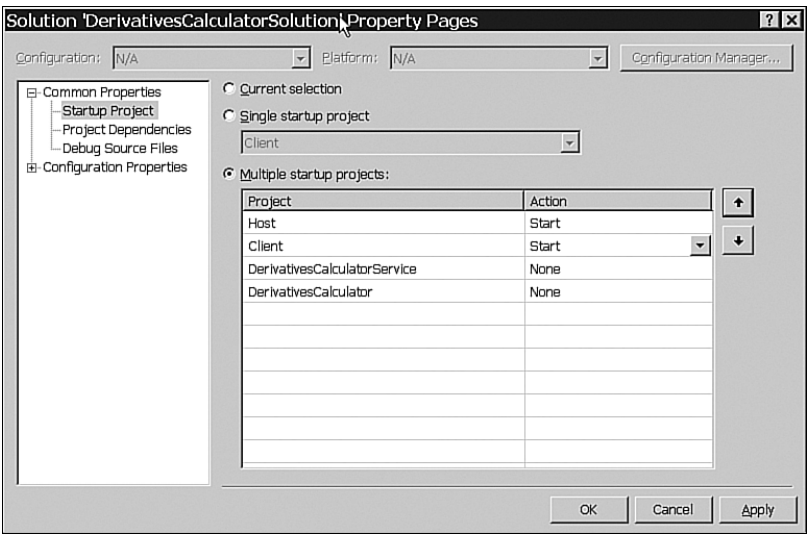


FIGURE 2.15 Startup project properties of the derivatives calculator solution.

- 11. Choose Debug, Start Debugging from the Visual Studio menus.
- 12. When there is activity in the console for the Host application, enter a keystroke into the console for the Client application. The client should obtain an estimate of the value of a derivative from the derivatives calculator service, as shown in Figure 2.16. Note that the value shown in the Client application console might vary from the value shown in Figure 2.16 due to variations in prevailing market conditions over time.
- 13. In Visual Studio, choose Debug, Stop Debugging from the menus.

Different Ways of Coding Windows Communication Clients

In the preceding steps for building a client for the derivatives calculator service, the code for the client was generated using the Windows Communication Foundation’s Service Metadata Tool. The generated code consists of a version of the `IDerivativesProxy` inter-

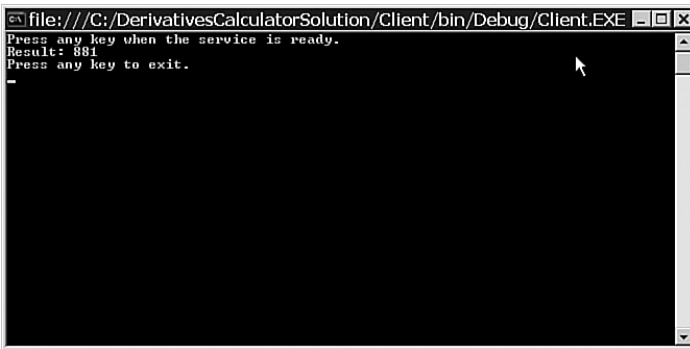


FIGURE 2.16 Using the derivatives calculator via a service.

face, and the code of a proxy class for communicating with the derivatives calculator service. The latter code is in Listing 2.6.

LISTING 2.6 Generated Proxy Class

```
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.CodeDom.Compiler.GeneratedCodeAttribute(
    "System.ServiceModel", "3.0.0.0")]
public partial class DerivativesCalculatorClient :
    System.ServiceModel.ClientBase<IDerivativesCalculator>,
    IDerivativesCalculator
{

    public DerivativesCalculatorClient()
    {
    }

    public DerivativesCalculatorClient(
        string endpointConfigurationName) :
        base(endpointConfigurationName)
    {
    }

    public DerivativesCalculatorClient(
        string endpointConfigurationName,
        string remoteAddress) :
        base(endpointConfigurationName, remoteAddress)
    {
    }
}
```