Expert Service-Oriented Architecture in C# 2005

Second Edition

Jeffrey Hasan with Mauricio Duran

Apress®

Expert Service-Oriented Architecture in C# 2005, Second Edition

ISBN-13 (pbk): 978-1-59059-701-9

ISBN-10 (pbk): 1-59059-701-X

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jonathan Hassell
Technical Reviewers: Mathew Upchurch, Omar Del Rio
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Richard Dal Porto
Copy Edit Manager: Nicole LeClerc
Copy Editors: Jennifer Whipple, Ami Knox
Assistant Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Dina Quan
Proofreader: Liz Welch
Indexer: Michael Brinkman
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 4

Design Patterns for Building Service-Oriented Web Services

Message-oriented Web services are the building blocks for service-oriented applications. In the previous chapter, you learned how message-oriented Web services are constructed, and what sets them apart from traditional RPC-style Web services. The main difference is that messages typically include complex types that are defined using custom XML schema files. Message-oriented Web services are effective at executing operations, whereby the input parameters feed into a process rather than dictating the process.

In contrast, procedure-style method calls are straightforward operations with a strong dependency on the input arguments. For example, the message-oriented StockTrader Web service provides a PlaceTrade operation that accepts the trade specifications, executes a complex trade operation, and then returns the details of the trade encapsulated in a complex data type (the Trade object). The simple input parameters trigger a complex operation and cause a complex type to be returned. There is no direct correlation between the input parameters and the complexity of the operation. In contrast, one example of a procedure-style Web method is a simple arithmetic Add operation that accepts two numeric input parameters. This Web method has nothing complicated happening internally, nor does it require that a complex data type be returned. What you get out of the method is directly correlated to what you send into it.

In this chapter, we need to make another conceptual leap, this time from message-oriented Web services to service-oriented Web services. Messages do not go away in this new architecture; they are just as important as ever. What is different is that Web services are not the central player in the architecture.

How to Build Service-Oriented Web Services

Service-oriented Web services act more as smart gateways for incoming service requests than as destinations in and of themselves. Let's revisit the complex SOA diagram from Chapter 1, reprinted here as Figure 4-1.

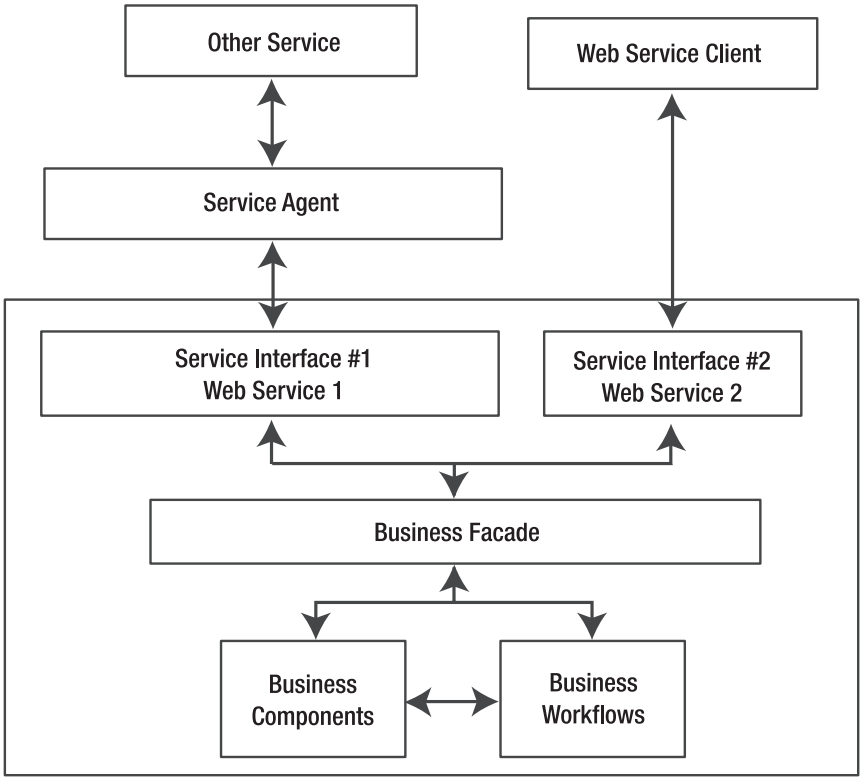

Figure 4-1. *Complex SOA*

Notice that Web services are not the ultimate endpoint destinations in this architecture. Instead, their purpose is to authenticate and authorize incoming service requests, and then to relay the request details to back-end business components and workflows for processing. This fact by no means diminishes the importance of their role; it just switches perspectives. Web services have certain unique properties that make them essential to this architecture:

- Web services process SOAP messages.
- Web services provide accessible (and discoverable) endpoints for service requests.
- Web services (optionally) authenticate and authorize incoming service requests. In this role they selectively filter incoming service requests and keep out unauthorized requests. (This feature is technically optional but it is an important available feature with WSE 3.0, and so is listed here as an essential property).

In contrast, other components in the architecture, such as the business components, do not have any of these properties. They do not expose publicly accessible endpoints. They

do not process SOAP requests directly. And they do not have the same ability to filter out incoming service requests based on security tokens. Note that business components can implement custom security checks through mechanisms such as code access security (CAS) and Active Directory checks, but these options are not comparable to the available mechanisms for Web services, which can accept encrypted and signed requests, and which inspect several aspects of the request directly, not just the identity of the caller.

So we have established that Web services play a unique role in SOA, one where they are an important support player rather than the ultimate destination endpoint. But what does this translate to in practical terms, and how is it different from before? The implication is that you need to build Web services differently to maximize the effectiveness of their role in SOA applications. This includes the following:

A renewed emphasis on breaking out Web service code-behind into separate class files and assemblies: This includes abstract IDC files (based on the applicable WSDL document). It also includes generating a dedicated assembly for encapsulating custom data type definitions (so that common data types may be used across multiple services and components using a common reference assembly).

Delegation of all business process logic to back-end business components: The Web service code-behind should be focused exclusively on preprocessing incoming request messages and then relaying the request details to the appropriate back-end business component. The Web service code-behind should not handle any business processing directly.

A focus on new kinds of service-oriented components: SOA architecture creates a need for different kinds of service components that may have no equivalent in other architectures. For example, SOA applications rely heavily on service agent components, which act as the middleman between separate Web services and which relay all communications between them. (You will learn how to build a service agent component in the section "Design and Build a Service Agent" later in this chapter.)

Be forewarned: some of the material in this chapter may strike you as unusual or unorthodox and certainly more complex than you are used to seeing with Web services development. This is not surprising given that SOA applications are still relatively new. Recall that it took several years for the *n*-tier architecture model to become fully formed and to gain wide acceptance as a standard. SOA will also go through an evolution. Some ideas will gain acceptance, while others will fall by the wayside. This chapter quite likely contains some of both, so read the chapter, absorb the material, and take with you as much or as little as you like.

The primary requirement that SOA imposes on a system is that its business functionality must be accessible through more than one type of interface and through more than one kind of transport protocol. Enterprise developers have long understood the need to separate out business functionality into a dedicated set of components. In Chapter 3, the StockTrader Web service implemented its business logic directly, based on an IDC file (defined in a separate, though embedded, class file). This approach is incorrect from an SOA perspective for two reasons:

Web services should not implement business logic directly in their methods: They should delegate this processing to dedicated business assemblies. This is because you cannot assume that the business logic will always be accessed through a Web service. What happens, for example, when a new requirement comes through asking you to implement an alternate interface that cannot or will not interact with a Web service? You need to have a separate, ready-to-use assembly for the business logic.

Web services and their associated WSDL documents should not be the original reference points for interface definitions: Certainly, a WSDL document must conform to an established interface definition, but it should not be establishing what that definition is. This information belongs in a dedicated reference assembly, and should be stored as an interface definition that can be implemented in different kinds of components.

The previous version of the StockTrader Web service is not compatible with SOA because it prevents common functionality from being accessible via multiple interfaces. To put it in blunt terms, the StockTrader Web service is simply incompatible with SOA because it is not abstract enough. What it needs to do instead is to act as a trusted interface to a back-end StockTrader business component. It cannot directly contain implementation for the StockTrader functions (such as getting quotes and placing trades). Instead, it must delegate this functionality to a back-end business component and focus on its primary role of authenticating and authorizing incoming service requests and then relaying these service requests to the back-end business component. In conjunction to this, the Web service is also responsible for relaying responses back to the client.

Consider another aspect to this architecture: type definitions. If you separate out common functionality across multiple components, how do they maintain a common understanding of type definitions? For example, how does every component maintain the same understanding of the Quote and Trade data types? XML Web services and their clients can share XSD schema information for custom data types via the service's published WSDL document. But this is not an efficient way to share type information between a middle-tier business component and a Web service, especially when the Web service is delegating requests to the business component. The more efficient approach is to generate a dedicated assembly that encapsulates the data type definitions as custom classes, and to include a reference to this assembly from wherever the custom data types are needed.

I have covered several challenging conceptual points, so now let's move on to code, and actually build a service-oriented Web service. Figure 4-2 is an architecture (and pseudo-UML diagram) that provides an alternate architecture for the original StockTrader Web service, one that will enable it to participate better in a larger SOA. Notice that the type definitions and interface definitions have been broken out into a separate assembly called StockTraderTypes, which is referenced by several components in the architecture.

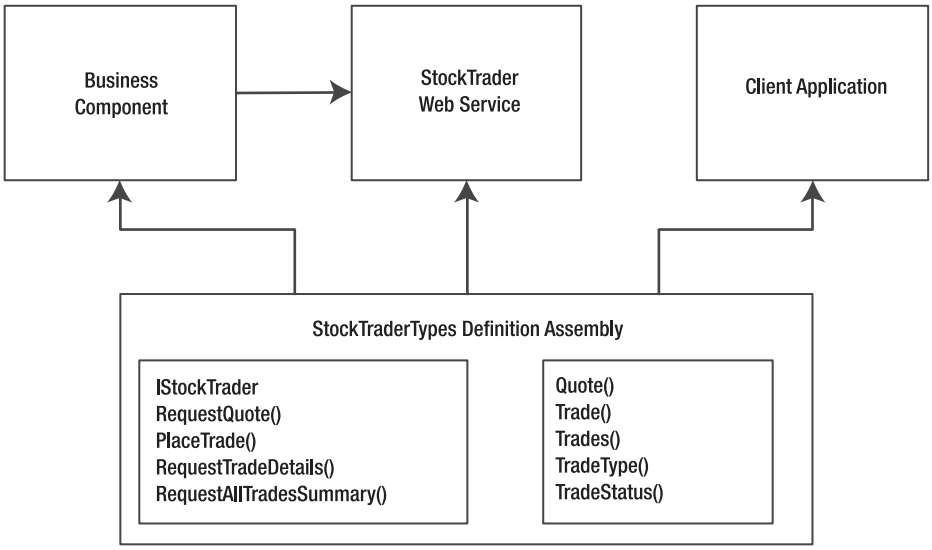

Figure 4-2. *Revised architecture for the StockTrader Web service showing how several components reference the common StockTraderTypes definition assembly*

Based on this UML diagram, there are six steps involved in building a message-oriented Web service that is compatible with SOA.

Step 1: Create a Dedicated Type Definition Assembly

Create a dedicated definition assembly for interfaces and type definitions. This assembly will be referenced by any component, service, or application that needs to use the interfaces or types.

Step 2: Create a Dedicated Business Assembly

Create a dedicated business assembly that implements logic for established interfaces and type definitions. This business assembly must reference the definition assembly from step 1. This ensures that the business assembly implements every available method definition.

Once this step is complete, you have the flexibility to build any kind of *n*-tier solution using the definition and business assemblies. This chapter focuses on building a service-oriented application that includes a Web service. But you could just as easily go a different route and build any kind of *n*-tier solution using the definition and business assemblies developed so far.

This point underscores the fact that in an SOA, Web services are simply a gateway to a set of methods and types that are controlled by other assemblies. The Web service itself merely provides a set of SOAP-enabled endpoints that are accessible over one or more transport protocols.

Step 3: Create the Web Service Based on the Type Definition Assembly

In the previous version of the StockTrader Web service, the definition information for the Web method implementations came from a dedicated IDC file, which provided abstract class definitions and class-based type definitions. But now this file is no longer needed because you have a dedicated definition assembly. The new Web service simply needs to import the definition assembly to have access to the required types and to the required interface.

Step 4: Implement the Business Interface in the Web Service

The Web service needs to import the business assembly so that it can delegate incoming service requests. Remember, the current architecture calls for a different level of abstraction, whereby the Web service itself does not control its interface, its data types, or the processing of business logic. Instead, it relies on other assemblies for this reference information and for this processing capability.

By implementing the interface, you ensure that you will not miss any methods because the project will not compile unless every interface method is implemented in the Web service. So, the definition assembly provides the interface definition, while the business assembly provides the processing capability for each method. All incoming Web service requests should be delegated to the business component, rather than implementing the business logic directly in the Web service.

The methods in this class file must be decorated with any required reflection attributes, such as WebMethod and SoapDocumentMethod. You always had to do this, so this is not new. But there is added importance now because many of these attributes will not be decorated elsewhere. Or if they are, they will not propagate to your class file. For example, the SoapDocumentMethod attributes are not included in the interface definition assembly (although the XML serialization attributes are). These attributes are not automatically carried over to the class file when it implements the definition assembly. As a matter of practice, we make sure that the interface definition assembly is decorated with the required serialization attributes, but we leave out attributes that relate to WebService and WebMethod attributes. This approach is implementation agnostic, meaning that it makes no assumptions about what kind of class file will implement the interface definition assembly.

■ **Note** Reflection attributes provide additional metadata for your code. The .NET runtime uses this metadata for executing the code. Class members are said to be decorated with attributes. Reflection attributes are a powerful tool because they enable the same code listing to be processed in different ways, depending on how it is decorated. Chapter 3 has a more complete discussion of reflection attributes, and Table 3-1 provides detailed property descriptions for the SoapDocumentMethod attribute.

Step 5: Generate a Web Service Proxy Class File Based on the WSDL Document

Proxy class files can still be generated directly from the Web service WSDL document, so this step does not have to change with a revised architecture in order to still work. However, the autogenerated proxy class file will not automatically utilize the dedicated definition assembly. This creates a significant issue because the proxy class file maintains its own type and interface definitions. Your goal is to have a central repository for this information. So in the interest of type fidelity, you need to modify the autogenerated proxy file to utilize the definition assembly rather than a separate copy of the same information.

Separate copies can be modified, and there is nothing to stop you from altering a proxy file so that it can no longer call the Web service it is intended for. This is why it is good to derive all types and interfaces from a common source.

Step 6: Create a Web Service Client

The Web service client uses the generated proxy class file from step 5 to set a reference to the new Web service. The client must also reference the type definition assembly from step 1, so that both the client and the Web service have a common understanding of the data types that are used by the Web services and its associated business assembly.

Some readers may see a red flag here because this approach creates a very tight coupling between the client and the Web service due to their mutual dependence on the same reference assembly. In contrast, it would be much easier to create a loosely coupled client that autogenerates a proxy file itself, using the Web service WSDL document. This autogenerated proxy file would include both methods and data types, so it would deviate from the more abstract approach that we are presenting here—namely, the approach of separating type definitions and method definitions into a dedicated assembly.

I am not advocating that you should always enforce this level of tight coupling between a Web service and its client. By definition, Web services are loosely coupled to their clients. This alternate approach is simply that—an alternate approach that can be implemented if the scenario is appropriate. In some cases, this approach will not even be feasible because the client may not have access to a dedicated assembly. But this approach may be warranted in other cases, particularly when you have a sensitive business workflow and you want to prevent any kind of miscommunication between a service and a client.

So, as with all the material in this book, absorb the information, consider the different approaches, but then decide which approach is most appropriate for your business requirements.

Design and Build a Service-Oriented Web Service

This section provides the information that you need to build a message-oriented Web service for use in an SOA. It is organized along the same six steps presented earlier and provides both conceptual information and implementation information.

Create the Definition Assembly (Step 1)

The definition assembly provides two important sets of information:

- Class definitions for all custom types that are exchanged in the system
- Interface definitions for each operation that the system supports

In this sense it is not unlike the autogenerated IDC file from Chapter 3. Recall that the type information in this file (StockTraderStub.cs) is autogenerated from an XSD schema file using the xsd.exe tool. The operations are manually inserted as abstract class methods that must be overridden by whatever class implements this file.

There are two differences between the definition assembly and the IDC file:

The operations are documented as interfaces rather than abstract class methods. This is because a given class can only derive from one other class at a time. Web service classes, for example, must derive either directly or indirectly from the System.Web.Services.WebService class. The Web service class cannot implement an additional interface unless it is provided as an invariant interface.

The definition assembly does not include Web service and SOAP-related attribute decorations. This is because it will be referenced from a variety of different assemblies, some of which have nothing to do with Web services. However, the definition assembly can still include XML serialization attributes.

Figure 4-3 shows a UML class diagram for the definition assembly. Notice the following two important points:

1. The type definitions are encapsulated in dedicated classes (e.g., Quote).
2. The method definitions are contained within an interface class called IStockTrader.

It is possible for a client project to reference the StockTraderTypes assembly solely for the purpose of accessing the custom data type definitions. The client does not need to implement the interface class, just because it is included in the assembly. But of course if they do, they will be required to implement every member of the interface.

To create the definition assembly, start by creating a new Class Library project in Visual Studio 2005 called StockTraderTypes, and add to it a single class file also called StockTraderTypes.

Listing 4-1 shows high-level pseudocode for the StockTraderTypes definition assembly.

Listing 4-1. *Pseudocode Listing for the StockTraderTypes Definition Assembly*

```
namespace StockTraderTypes
{
    public interface IStockTrader {}
    public class Quote {}
    public class Trade {}
    public class Trades {}
    public enum TradeStatus {}
    public enum TradeTypes {}
}
```

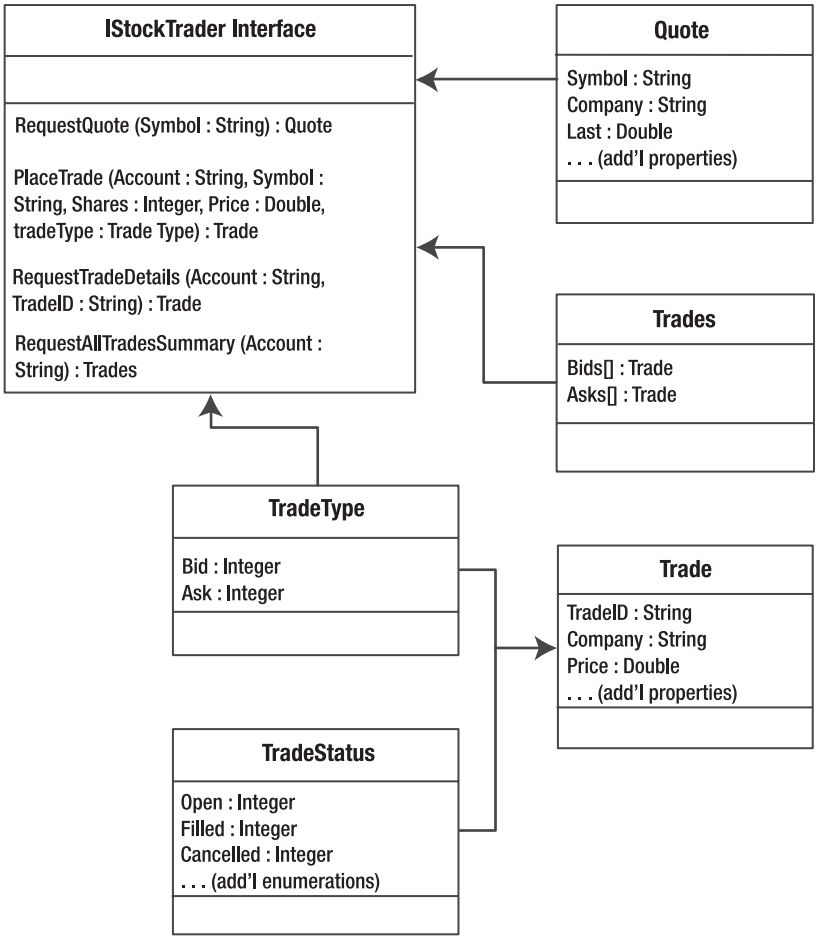

Figure 4-3. *UML class diagram for the StockTraderTypes definition assembly*

Listing 4-2 presents a more detailed code listing, excluding XML serialization attributes. These attributes are important because they directly relate the code elements to XML elements in the associated XSD schema (which is assigned to a qualified namespace at `http://www.bluestonepartners.com/schemas/StockTrader/`).

Listing 4-2. *Detailed Code Listing for the StockTraderTypes Definition Assembly*

```
using System;
using System.Xml.Serialization;

namespace StockTraderTypes
 {
  public interface IStockTrader
  {
    Quote RequestQuote(string Symbol);
```

```
    Trade PlaceTrade(string Account, string Symbol, int Shares, ➥
        System.Double   Price, TradeType tradeType);
    Trade RequestTradeDetails(string Account, string TradeID);
    Trades RequestAllTradesSummary(string Account);
  }

  public class Quote
  {
    public string Symbol;
    public string Company; // Additional type members not shown
  }

  public class Trade
  {
    public string TradeID;
    public string Symbol; // Additional type members not shown
  }

  public class Trades
  {
    public string Account;
    public Trade[] Bids;
    public Trade[] Asks;
  }

  public enum TradeStatus
  {
    Ordered,
    Filled, // Additional type members not shown
  }

  public enum TradeType
  {
    Bid,
    Ask
  }

}
```

This is all the work that is required to create a definition assembly that can be reused across other components, services, and applications.

Create the Business Assembly (Step 2)

The business assembly implements the IStockTrader interface that is defined in the StockTraderTypes definition assembly. This logic was previously implemented directly in the Web service class file. But this design is very limiting because it isolates the business logic inside a

specialized class file. The business assembly provides a standard middle-tier component that can be referenced and invoked by a wide variety of consumers, not just Web services.

Creating the business assembly requires three steps:

1. Create a new Class Library project in Visual Studio 2005 called StockTraderBusiness, and add to it a single class file also called StockTraderBusiness.
2. Set a reference to the StockTraderTypes assembly. For now you can create all projects in the same solution, and then set a reference to the StockTraderTypes project (from the Projects tab in the Add Reference dialog box).
3. Import the StockTraderTypes namespace into the StockTraderBusiness class file and implement the IStockTrader class. Implement code for each of the interface operations. You will get compiler errors if you attempt to build the solution without implementing all of the operations.

Listing 4-3 displays the pseudocode listing for the StockTraderBusiness business assembly.

Listing 4-3. *Pseudocode Listing for the StockTraderBusiness Business Assembly*

```
using System;
using StockTraderTypes;

namespace StockTraderBusiness
{

  public class StockTraderBusiness : StockTraderTypes.IStockTrader
  {
    public Quote RequestQuote(string Symbol)
    {
         // Implementation code not shown
    }
    public Trade PlaceTrade(string Account, string Symbol, int Shares, ➥
        System.Double Price, TradeType tradeType)
    {
         // Implementation code not shown
    }
    public Trade RequestTradeDetails(string Account, string TradeID)
    {
         // Implementation code not shown
    }
    public Trades RequestAllTradesSummary(string Account)
    {
         // Implementation code not shown
    }
  }

}
```

The business assembly is the sole location for implemented business logic and the final destination for incoming service requests. The previous listing looks very spare because it does not show the implementation code for any of the methods. You can refer to the sample project to view the full code listing. Very little implementation code is shown in this chapter because it is of secondary importance. It is more important that you feel comfortable with the interfaces and the architecture of the components.

Create the Web Service (Steps 3–5)

The previous version of the StockTrader Web service implemented an IDC file for operations and types. This file is no longer needed because the same information is now provided by the definition assembly.

Create a new Web service project named StockTraderContracts in the Visual Studio 2005 solution, and rename the .asmx file to StockTraderContracts. Use the Add Reference dialog box to set references to the StockTraderBusiness and StockTraderTypes assemblies.

Listing 4-4 displays the pseudocode listing for the StockTraderContracts Web service.

Listing 4-4. *Pseudocode Listing for the StockTraderContracts Web Service*

```
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Web.Services.Description;

using StockTraderTypes;
using StockTraderBusiness;

namespace StockTrader
{

    public class StockTrader : System.Web.Services.WebService, ➥
        StockTraderTypes.IStockTrader
    {

    [WebMethod]
    [SoapDocumentMethod(RequestNamespace= ➥
        "http://www.bluestonepartners.com/schemas/StockTrader/",
        ResponseNamespace="http://www.bluestonepartners.com/schemas/StockTrader/",
        Use=SoapBindingUse.Literal, ParameterStyle=SoapParameterStyle.Bare)]
    [return: System.Xml.Serialization.XmlElement("Quote", Namespace=
        "http://www.bluestonepartners.com/schemas/StockTrader/")]
    public Quote RequestQuote(string Symbol)
    {
        // Implementation code not shown
    }
     [WebMethod]
    //XML and SOAP serialization attributes not shown
    public Trade PlaceTrade(string Account, string Symbol, int Shares, ➥
```

```
        System.Double Price, TradeType tradeType)
    {
        // Implementation code not shown
    }
     [WebMethod]
    //XML and SOAP serialization attributes not shown
    public Trade RequestTradeDetails(string Account, string TradeID)
    {
        // Implementation code not shown
    }
     [WebMethod]
    //XML and SOAP serialization attributes not shown
    public Trades RequestAllTradesSummary(string Account)
    {
        // Implementation code not shown
    }

    }

}
```

The Web service methods no longer implement their own business logic. Instead, every method must delegate incoming requests to the business assembly. For example, Listing 4-5 shows how the RequestQuote Web method delegates an incoming service request to the RequestQuote method in the business assembly.

Listing 4-5. *Delegation in the RequestQuote Web Method*

```
[WebMethod]
// XML and SOAP attributes not shown
public Quote RequestQuote(string Symbol)
{
    StockTraderBusiness b = new StockTraderBusiness();
    Quote q = b.RequestQuote(Symbol);
    return q;
}
```

The code is extremely simple because the Web service and the business assembly share the same type definitions and implement the same interface. The communication between the parties is seamless because they share a common vocabulary.

Figure 4-4 shows the Solution Explorer window for the project, with the References nodes expanded so that you can see how the assembly references are configured in each of the projects: StockTraderTypes, StockTraderBusiness, and StockTraderContracts. In addition, this figure includes the client console application, StockTraderConsole, which is described in step 6.

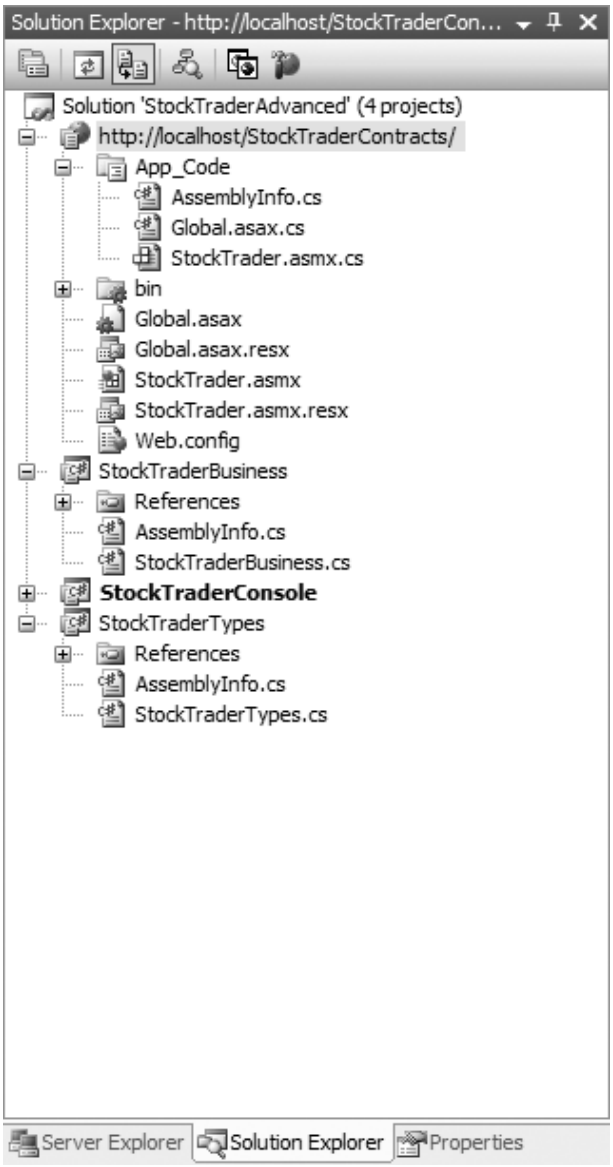

Figure 4-4. *The Solution Explorer view for the StockTraderAdvanced solution*

Create the Web Service Client (Step 6)

In this example, you are going to see how to build a tightly coupled Web service client that references the same definition assembly as the Web service itself. But as we clarified earlier, it is often advisable to implement a loosely coupled Web service client, whereby the client generates its own proxy file based on the Web service WSDL document and its associated XSD schemas. In fact, SOA promotes loose coupling between Web services and consumers.

As we stated earlier, our purpose in building a tightly coupled Web service client is to show you an alternate approach to building clients. In some cases, you will want to build a tightly coupled Web service client in order to prevent any miscommunication or misunderstanding between the Web service and its client as to what methods and types are supported. Certainly, type definitions can change, and so tight coupling can add an additional burden to the developer of the client. However, WSDL definitions can also change just as easily, and there is no clear way for a Web service to communicate interface changes to its clients.

Ultimately, we advocate the design approach of loose coupling between a Web service and its clients. The alternative tightly coupled approach that we are presenting here simply has the Web service itself referencing a type definition assembly and delegating all of its business logic to a dedicated business assembly. Technically, this is tight coupling between the Web service and client, as opposed to the traditional loose coupling between client and service, where the proxy class is generated as needed based on the current Web service WSDL specification. The material in this chapter provides everything you need to understand and implement both loosely coupled and tightly coupled designs. We will look at both approaches next.

Build a Loosely Coupled Web Service Client

Add a new console application named StockTraderConsole to the Visual Studio 2005 solution, and then do one of the following:

- Generate the proxy class manually with the wsdl.exe command-line utility applied to the Web service WSDL document.
- Use the Add Reference wizard in Visual Studio 2005 to automatically generate the proxy class in the client project.

Once you have generated the proxy class, you simply reference it directly from the client code, as shown in Listing 4-6.

Listing 4-6. *Web Service Consumer Code*

```
// Create an instance of the Web service proxy
StockTraderProxy serviceProxy = new StockTraderProxy();

// Retrieve the Web Service URI from app.config
serviceProxy.Url = ConfigurationSettings.AppSettings["remoteHost"];

// Call the Web service to request a quote
Quote q = serviceProxy.RequestQuote("MSFT");

// Display the Quote results in the form
Console.WriteLn("\t:Company:\t " + q.Company);
Console.WriteLn("\t:Symbol:\t " + q.Symbol);
Console.WriteLn("\t:Last:\t " + q.Last.ToString());
Console.WriteLn("\t:Prev Close:\t " + q.Previous_Close.ToString());
```

For more information on building loosely coupled clients, please refer to Chapter 3.

Build a Tightly Coupled Web Service Client

Autogenerated proxy class files are completely self-contained and essentially provide the client with a separate local copy of the interface and type definitions that the Web service supports. If the Web service interface changes, the client will not automatically pick up on these changes unless they clear the existing Web reference and regenerate the proxy class. You can manage this risk by modifying the autogenerated proxy class to conform to the standard interface and type definitions that are contained in the StockTraderTypes assembly.

Add a new console application project named StockTraderConsole to the Visual Studio 2005 solution file and copy over the proxy class file from the previous chapter's StockTrader Web service. Alternatively, you can autogenerate the proxy class from within the StockTraderConsole project as follows:

Step 1: Use the Add Web Reference Wizard to autogenerate the proxy class for the StockTraderContracts Web service at `http://localhost/StockTraderContracts/StockTrader.asmx`.

Step 2: The autogenerated proxy class file is called Reference.cs and is stored in the solution under the Web References\[Reference Name]\Reference.map subproject folder. (If you do not see this file, you can use the Project ➤ Show All Files menu option to expand all files.)

Step 3: Open the Reference.cs file and copy the entire code listing over to a new C# class file called StockConsoleProxy.cs.

Rename the proxy class file to StockConsoleProxy, and then do the following:

Step 1: Add a reference from the StockTraderConsole project to the StockTraderTypes assembly.

Step 2: In the StockConsoleProxy class, import the StockTraderTypes namespace and add the IStockTrader interface to the StockConsoleProxy interface list immediately following SoapHttpClientProtocol.

Step 3: Comment out all of the type definitions in the StockConsoleProxy class. These include Quote, Trade, Trades, TradeType, and TradeStatus. They are now redundant because the definition assembly contains the same type definitions.

The pseudocode for the proxy class now reads as shown in Listing 4-7 (modifications from the previous, or autogenerated, proxy classes are shown in bold).

Listing 4-7. *The Proxy Class for the StockTraderContracts Web Service, Modified to Reference the Type Definition Assembly StockTraderTypes*

```
using System.Web.Services;
using System.Web.Services.Protocols;

using StockTraderTypes;
```

```
[System.Web.Services.WebServiceBindingAttribute(Name="StockTraderServiceSoap",
    Namespace="http://www.bluestonepartners.com/schemas/StockTrader")]
public class StockConsoleProxy : SoapHttpClientProtocol, ➥
    StockTraderTypes.IStockTrader
{

    // Pseudo-code only: implementations and attributes are not shown
    public Quote RequestQuote() {}
    public System.IAsyncResult BeginRequestQuote() {}
    public System.IAsyncResult EndRequestQuote() {}

    // Additional operations are not shown
    // These include PlaceTrade(), RequestTradeDetails(),
    // and RequestAllTradesSummary()

    // Type definitions are commented out of the proxy class
    // because they are redundant to the type definition assembly
    // These include Quote, Trade, Trades, TradeType, and TradeStatus

}
```

These are trivial modifications because the proxy class already implements all of the IStockTrader interface members. The benefit of explicitly adding the IStockTrader interface is to ensure that the proxy class remains constrained in the way it implements the StockTrader operations. You could modify the proxy class in many other ways, but as long as the StockTrader operations remain untouched (interfacewise at least), the client application will compile successfully.

Once the proxy class has been modified, the client code can be implemented in the console application. The StockTraderTypes namespace must be imported into the client class file so that the client can make sense of the type definitions. No additional steps are required to use the definitions assembly. Listing 4-8 shows the client code listing for calling the RequestQuote operation.

Listing 4-8. *Client Code Listing for Calling the RequestQuote Operation*

```
using StockTraderTypes;

namespace StockTraderConsole2
{
class StockTraderConsole2
{

[STAThread]
static void Main(string[] args)
{
StockTraderConsole2 client = new StockTraderConsole2();
client.Run();
}
```

```
public void Run()
{
// Create an instance of the Web service proxy
StockConsoleProxy serviceProxy = new StockConsoleProxy();

// Configure the proxy
serviceProxy.Url = ConfigurationSettings.AppSettings["remoteHost"];

// Submit the request to the service
Console.WriteLine("Calling {0}", serviceProxy.Url);
string Symbol = "MSFT";
Quote q = serviceProxy.RequestQuote(Symbol);

// Display the response
Console.WriteLine("Web Service Response:");
Console.WriteLine("");
Console.WriteLine( "\tSymbol:\t\t" + q.Symbol );
Console.WriteLine( "\tCompany:\t" + q.Company );
Console.WriteLine( "\tLast Price:\t" + q.Last );
Console.WriteLine( "\tPrevious Close:\t" + q.Previous_Close );
}

}
}
```

Figure 4-5 displays a client console application that interfaces to the StockTraderContracts Web service using the modified proxy class. Please refer to the sample application (available from the Source Code/Download section of the Apress web site at `http://www.apress.com`) for full code listings.

Figure 4-5. *Client console application for the StockTraderContracts Web service*

This concludes the overview of how to build a tightly coupled Web service client. Again, we would like to emphasize that this approach is not consistent with a pure SOA environment where the clients remain completely decoupled from the Web services they consume. However, it is always useful to consider alternative approaches and to realize new possibilities even if they never make it into a production environment.

Next, we will discuss a type of component that is unique to the service-oriented environment: the service agent.

Design and Build a Service Agent

Service agent components are essentially translator components that act as the intermediary between a business component and an external Web service. By *external*, we mean external to the domain where the business object is located. Service agents were discussed in some detail in Chapter 1 and are included in Figure 4-1 in this chapter. Briefly, the purpose of a service agent is to eliminate complexity in a business component by managing all interactions with an external service. If service agents did not exist, the business component would need to implement proxy classes and all of the associated error handling logic for working with external services. Clearly, this adds an undesirable layer of code and complexity to the business component that is superfluous because the business client will never call this code directly.

For example, consider Company A, which has built a business component that processes stock trades and provides stock quotes. In order to provide this functionality, the business component uses an external Web service that is provided by a premier brokerage company, Company B. Company A uses its own custom data types, which are encapsulated in the StockTraderTypes assembly. Company B, however, defines its own data types that are equivalent but not the same as Company A's. For example, Company A uses a Quote data type that defines a property called Open, for the day's opening share price. Company B uses a Quote data type that defines an equivalent property called Open_Ext. Company A uses strings for all of its custom data type properties, whereas Company B uses a mix of strings, floats, and dates.

Given these differences, Company A's service agent will perform two important functions:

1. It will implement the infrastructure that is required to communicate with Company B's external service. It will be responsible for the maintenance work that will be required if the external service updates its interface.

2. It will translate the responses from the external service and will relay them back to Company A's business component using a mutually understood interface.

The benefits of a service agent are clear: the service agent eliminates complexity for Service A's business component because it encapsulates all of the implementation details for interacting with the Web service and relays the requests back in the format that the business component wants. Figure 4-6 provides a schematic representation of this architecture.

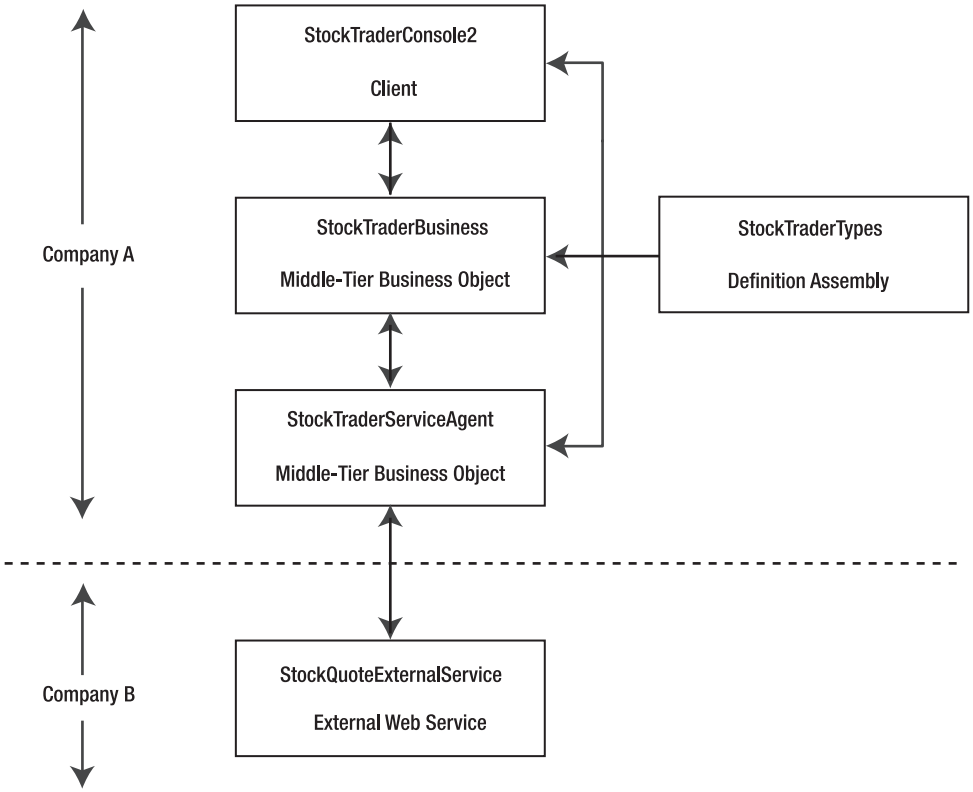

Figure 4-6. *SOA with a service agent*

Now let's look at how you implement this architecture in code.

Implement the StockTrader SOA Application Using a Service Agent

The StockTrader Web service has evolved in this chapter to where it delegates all requests to a business assembly (StockTraderBusiness). If a client contacts the Web service to request a stock quote, the Web service delegates the request to the business object's RequestQuote method. The Web service does not know or care how this method returns a stock quote, but it does expect to receive one every time it makes a request.

For the next evolution of the StockTrader Web service, your company signs a partnership agreement with another company that is a premier provider of stock quotes. You decide that going forward the StockTraderBusiness assembly will delegate all stock quote requests to this external service. The StockTrader Web service will continue to delegate requests to the business assembly, but the business assembly, in turn, will delegate the requests again, this time to an external Web service. You decide to build a service agent to minimize any change to the business assembly. Figure 4-7 shows the Solution Explorer for the solution that you are going to build, with selective References nodes expanded so you can see the relationships between the different components.

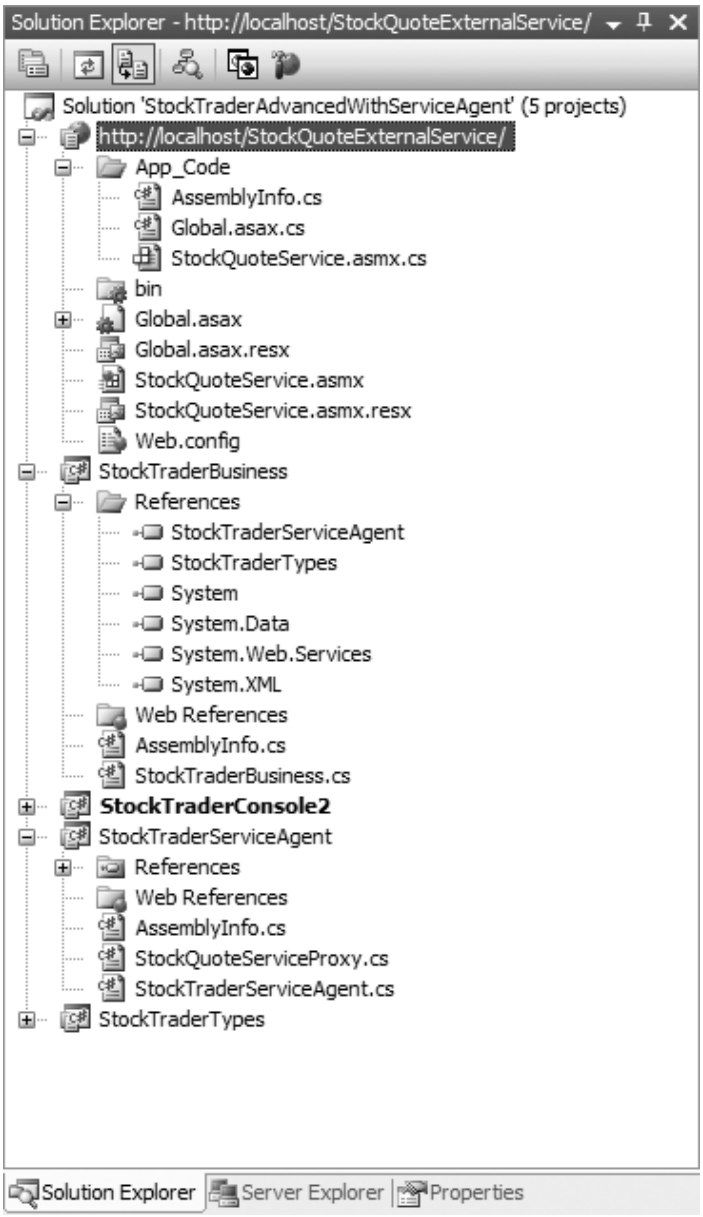

Figure 4-7. *Solution Explorer for the StockTrader SOA application, including a service agent*

The five components in this application are as follows:

1. *StockTraderConsole2*: The client application, providing a user interface
2. *StockTraderBusiness*: The middle-tier business component that handles processing for the client

3. *StockTraderServiceAgent*: The service agent used by the business component for communicating with external services
4. *StockTraderTypes*: The common type definition assembly, which is referenced by the three preceding components
5. *StockQuoteExternalService*: The external Web service

If this gets confusing, you can consult either Figure 4-6 or Figure 4-7, which include all five of these components. Let's look at how to build each component in turn, going from bottom to top, in the order of the service request workflow, starting with the external StockQuoteExternalService Web service.

The External Web Service (StockQuoteExternalService)

StockQuoteExternalService is a simple Web service that provides a single Web method for requesting stock quotes (RequestQuoteExt), and it returns its own equivalent to the StockTraderTypes.Quote type, which is named QuoteExt. The Quote and QuoteExt types are equivalent, but they differ from each other in three ways:

1. The QuoteExt type conforms to a different qualified namespace from the Quote type. Each type conforms to its own XSD schema file.
2. The QuoteExt type does not contain equivalents to the Quote type's Change and Percent_Change properties.
3. The QuoteExt type provides a time stamp property named DateTime_Ext, which is of type System.DateTime. The Quote type provides an equivalent time stamp property named DateTime that is of type String.

These are admittedly minor differences, but they illustrate the point. When you call an external service, it is unlikely that their type definitions will be equivalent to yours. You have to be prepared for some manual work to translate the differences.

In real life, of course, you would not have to create the external service yourself, but for the purposes of this demonstration you do.

The Service Agent (StockTraderServiceAgent)

The service agent implements the same interface and type definitions as the business assembly by referencing the StockTraderTypes assembly (as shown in Figure 4-6). The service agent also includes a proxy class for the StockQuoteExternalService external Web service.

Listing 4-9 shows the code listing for the service agent, including the complete listing for its RequestQuote method.

Listing 4-9. *The StockTraderServiceAgent Code Listing*

```
using System;
using StockTraderTypes;

namespace StockTraderServiceAgent
{
    public class StockTraderServiceAgent : StockTraderTypes.IStockTrader
    {
    public StockTraderServiceAgent(){}

    public Quote RequestQuote(string Symbol)
    {
        Quote q = null;

        // Request a Quote from the external service
        QuoteExt qe;
        StockQuoteService serviceProxy = new StockQuoteService();
        qe = serviceProxy.RequestQuoteExt("MSFT");

        // Create a local Quote object (from the StockTraderTypes namespace)
        q = new Quote();

        // Map the external QuoteExt object to the local Quote object
        // This requires some manual work because the types
        // do not map exactly to each other
        q.Symbol = Symbol;
        q.Company = qe.Company_Ext;
        q.DateTime = qe.DateTime_Ext.ToString("mm/dd/yyyy hh:mm:ss");
        q.High = qe.High_Ext;
        q.Low = qe.Low_Ext;
        q.Open = qe.Open_Ext;
        q.Last = qe.Last_Ext;
        q.Previous_Close = qe.Previous_Close_Ext;
        q.Change = (qe.Last_Ext - qe.Open_Ext);
        q.PercentChange = q.Change/q.Last;
        q.High_52_Week = qe.High_52_Week_Ext;
        q.Low_52_Week = qe.Low_52_Week_Ext;

        return q;
    }

    public Trade PlaceTrade(string Account, string Symbol, int Shares, ➥
        Double Price, TradeType tradeType)
    {
        // Implementation not shown
    }
```

```
    public Trades RequestAllTradesSummary(string Account)
    {
        // Implementation not shown
    }

    public Trade RequestTradeDetails(string Account, string TradeID)
    {
        // Implementation not shown
    }

    }
}
```

The code listing is very straightforward and shows how the service agent delegates its RequestQuote method to the external service's RequestQuoteExt method. The service agent performs some manual translations to map between its native Quote type and the external QuoteExt type. Finally, the agent returns a native Quote object to the consuming application, which in this case is the business assembly.

The Business Assembly (StockTraderBusiness)

The business component sets references to both the service agent assembly and the definition assembly of custom types. Listing 4-10 shows how the business component calls the service agent.

Listing 4-10. *The StockTrader Business Component Calling the Service Agent*

```
using System;
using StockTraderTypes;
using StockTraderServiceAgent;

namespace StockTraderBusiness
{
    public class StockTraderBusiness : StockTraderTypes.IStockTrader
    {
        public StockTraderBusiness() {}

    public Quote RequestQuote(string Symbol)
    {

        // Create a new Quote object
        Quote q = new Quote();

        // Call the service agent
        StockTraderServiceAgent sa = new StockTraderServiceAgent();
        q = sa.RequestQuote(Symbol);
```

```
        return q;
    }

    }
}
```

As you would expect, the listing is very simple because the business assembly no longer has to provide its own implementation of the Quote request logic.

In summary, service agents are an elegant solution when you need to interface with one or more external services and wish to isolate the code that handles the communication. Service agents provide stability to a business assembly by bearing the responsibility of ensuring successful calls to external services and returning results in a form that the business assembly natively understands. Service agents can also act as intermediaries between two or more Web services.

This concludes the discussion of how to build basic service-oriented Web services.

Summary

In this chapter, we expanded on the previous discussion of message-oriented Web services and showed you a six-step process for designing and building a service-oriented Web service from scratch:

Step 1: Create a dedicated type definition assembly.

Step 2: Create a dedicated business assembly.

Step 3: Create the Web service using the type definition assembly.

Step 4: Implement the business interface in the Web service.

Step 5: Delegate processing logic to the business assembly.

Step 6: Create a Web service client.

You saw how to build both tightly coupled clients and loosely coupled clients. In most SOA applications you will want to build loosely coupled clients, but under some circumstances you may want a higher level of control over the type definitions. Tightly coupled clients reference the same type definition as the assembly rather than generating their own using a proxy class.

Finally, we discussed the service agent component, which is a special feature of service-oriented applications. The service agent manages communication between a business assembly and an external Web service. It can also act as the intermediary between two or more Web services.

The goal of this chapter is to help you rethink your approach to Web services design so that you can start thinking in terms of SOA.