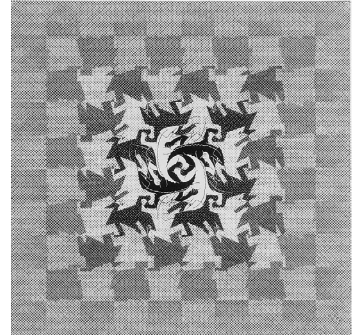
FOR PUBLIC RELEASE



2

Framework Patterns: Exception Handling, Logging, and Tracing

OVERVIEW

One of the primary success factors for implementing a robust framework is providing enough information at runtime so that the system can be effectively monitored. Despite how good a programmer you think you are, there will always be bugs. Hopefully, by the time your application goes to production, most of these can be discovered. However, no matter how diligent you may be, problems still arise. Eventually, you will be faced with solving the issue as quickly and as painlessly as possible. The better your framework is built to handle both preproduction and postproduction problems, the faster they will be solved.

Information is key to solving any problem. The more accurate, abundant, and accessible the errata, the better. This is especially true in postproduction, when

you may not have the liberty of being seated at the desktop of your users trying to solve their issues. I hope the topics in this chapter will provide you with some "best practices" that you can employ in any application, typically at the middle tier. I provide a melee of ideas, dos, and don'ts here to give you a few tips on which can be incorporated at any level. Unlike design patterns, these are not fixed to any one design but are more specific to the .NET framework. For those repeatable implementation steps, I've included a few implementation patterns that will add some benefit to any new or existing middle tier.

Like the remaining the chapters in this book, this chapter serves as a cookbook of ideas and does not have to be read in any particular order. The following topics will be covered in this chapter:

- Exception Handling
- Exception Logging
- Exception Chaining
- Building a Base Exception Class
- Tracing and Trace Listening
- Error and Call Stacks
- When, Where, and How to Log
- SOAP Exceptions and SOAP Faults
- Interop Exception Handling

The following implementation patterns will be described in detail:

- Remote Tracer
- Custom SOAP Exception Handler
- System Exception Wrapper
- SOAP Fault Builder

EXCEPTION HANDLING

Those already familiar with Java exception handling should feel quite comfortable with the .NET implementation. In fact, if you are reading this book, you should already be throwing and catching exceptions in your code. However, I would like to go over some “advanced-basics” in the area of exception handling before we move on to creating a “base exception” class (sprinkled with a few implementation patterns). Error, or exception handling is one of those things that everyone does but unfortunately, few do it very well. With robust handling, logging, and display, your applications will better stand the test of time and should save you hours in product support. From here forward, I use the word *exception* in place of *error*, and vice versa; neither is meant to be distinctive of the other. I assume throughout this chapter that you have already had some experience working with *structured exception handling* (SEH) because this will not be a tutorial on its basics.

Figure 2.1 illustrates a typical exception-handling scenario. This should provide you generally with the proper flow of throwing and catching exceptions. You’ll notice that once an exception is deemed *unrecoverable*, information is then added to provide the system with as much information as possible for determining the problem. This is where a *base exception* class becomes useful. Typically, the more you can centralize *value-added* features to this class, the easier it will be to integrate and leverage from within your framework. This is especially true of large development teams. How many times have there been framework features that were useful but unused? By centralizing important functionality such as logging in your base class and by standardizing how the exceptions should be thrown, you help guarantee its proper use. This also takes any guesswork out of proper exception handling for those developers using your framework. In an upcoming section, I will provide some of the steps for creating such a base class.

Application-Specific Exceptions

Like its C++ and Java predecessors, .NET uses a base *System.Exception* class as the parent of all thrown exceptions. Many applications can simply use this base class for throwing errors and are not required to implement their own. You may also wish to stick with using the System.SystemException-derived exceptions that come with the FCL. These include exception classes such as *System.Web.Ser-*

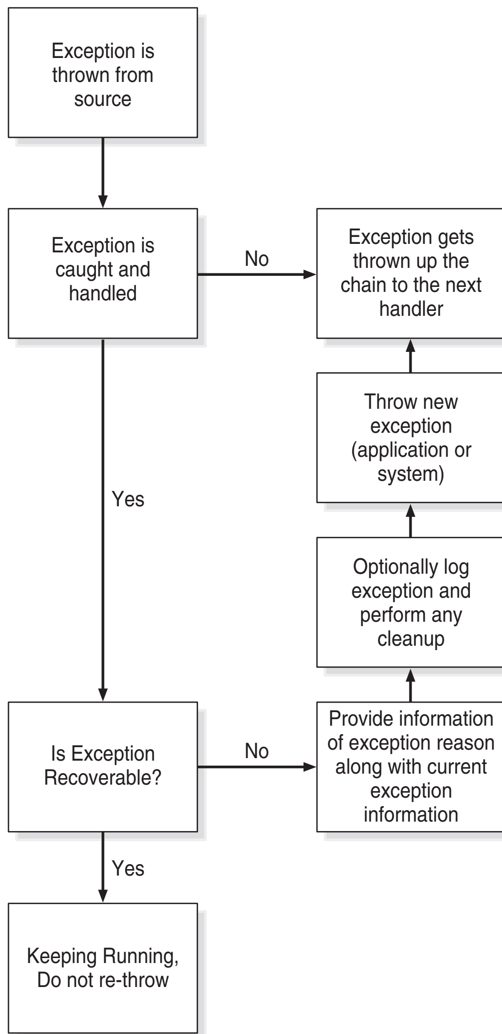


Figure 2.1: Exception flow: How most exceptions should be handled.

vices.Protocols.SoapException or *System.IO.IOException*. System exceptions of these types will also be used throughout the samples in this book. However, for special handling scenarios or to enhance your architecture, it is advisable to build your own exception base class. From there, you can begin to add functionality common to all exception handling for your framework.

From a base exception class, you can create specific error-handling classes and further centralize the process by which errors should be handled, displayed, and

recorded in your architecture. This includes general framework exceptions, such as the sample *FtpException* class you will see shortly, but it also includes all those business rule-specific scenarios that warrant special handling. All custom application exceptions, including your own framework exceptions, should derive from System.ApplicationException and not System.Exception directly. Even though System.ApplicationException directly derives from System.Exception, they have identical features. However, it is important to make the distinction between the two when designing your exception classes.

Rather than using method return codes such as the HRESULTS that we have come to know and love with COM, .NET is a type-based system that relies on the exception type to help identify errors. Throwing the appropriate exception type during a given scenario is as important as specifying the correct HRESULT in the world of COM. This doesn't mean we forgo all error codes. It simply means that leveraging the exception types will initiate a more robust error-handling system.

An example of a feature-specific exception handling class can be found when providing FTP services in your architecture (as shown below). Later, I will describe how this is used when we talk about creating protocol-specific requesters in the next chapter. I am just using this as a simple example, so bear with me. Here, the exception class is expanded to facilitate FTP-specific errors and the protocol return codes that are used to describe the error. The code displayed simply delegates most of the handling to the base class *BaseException*. Our BaseException class will be described shortly. The point here is that I can now better control how FTP errors should be handled.

In this example, I am simply displaying the error in a specific format when the Message property is used. Building function-specific exception classes also eliminates the guesswork for developers having to utilize an existing framework: If a specific exception class is provided, use it. If certain rules dictate even a future possibility of special error processing, error display, or error recording, then building a custom exception class would be prudent. Once the base exception class is built for a framework, function-specific exception classes can be driven from a template, such as the following:

Listing 2.1: A sample BaseException child class.

```
public class FtpException : BaseException
{
   /// <summary>
   /// Message for the ftp error
   /// </summary>
   private string m_sMessage;

   public FtpException(){;}

   public FtpException(string sMessage, bool bLog) :
         base(sMessage, bLog){;}

   public FtpException(string sMessage, System.Exception
      oInnerException, bool bLog) : base(sMessage, oInnerException,
         bLog){;}

   public FtpException(object oSource, int nCode, string sMessage,
        bool bLog) : base(oSource, nCode, sMessage,      bLog){;}

   public PMFtpException(object oSource, int nCode, string sMessage,
         System.Exception oInnerException, bool bLog) :
         base(oSource, nCode, sMessage, oInnerException, bLog){;}

   public new string Message
   {
         get
         {
               int nCode = base.Code;
               string sMessage = GetFtpMessage(nCode);
               return new StringBuilder(
                                 "FTP Server stated:[")
                                 .Append(nCode)
                                 .Append(" ")
                                 .Append(sMessage)
                                 .Append("]")
                                 .ToString();
         }
         set {m_sMessage = value;}
   }
```

The Listing 2.1 example is not all that different from a regular exception, other than two things. We use this FtpException class to display our FTP error in a specialized way. With the following helper function, I am using the returned FTP code to look up the corresponding FTP error message from a string table called *FTP-Messages* through the System.Resources.ResourceManager. Once the code descrip-

tion is returned, I then build the custom FTP error message, using both the code and the description.

Listing 2.2: One example of a "value-added" operation on a base exception child.

```
/// <summary>
/// Helper to return messages from ftp message resource string table
/// </summary>
/// <param name="sCode"></param>
/// <returns></returns>
public static string GetFtpMessage(int nCode)
{
   FtpMessages = new ResourceManager("FTPMessages",
      Assembly.GetExecutingAssembly());
   FtpMessages.GetString(new StringBuilder("Code_")
                           .Append(nCode).ToString());
}
```

The other differentiator between my FtpException class and any other *System.ApplicationException* class is the constructors for this class. Notice that most of the constructors delegate to the BaseException class. This is where most of the exception handling is done. Features handled by the BaseException class include some of the following, which I will talk about in more detail in this chapter:

- Automated Exception Chaining—Handles each exception as it gets passed back up to the caller of the method containing the exception. As each exception is thrown, the cause of the exception is passed as the "inner exception" thus becoming part of the newly thrown error. This continues up the chain until the information gathered can be used for tracking the original error, logging, or handling in some other fashion.
- Automated Error Stack Formatting and Display—While exceptions are chained, they can be tracked by building an error stack containing each chained exception. Typically, this is accomplished by capturing the important elements during chaining and building a string that can be displayed or made durable.
- Automated Call Stack Formatting and Display—This is identical to that of building an error stack exception. The .NET framework automates this by providing an error stack property that can be accessed at any time. The error stack will provide details as to the method and line number causing the error. This information can also be made durable or displayed.

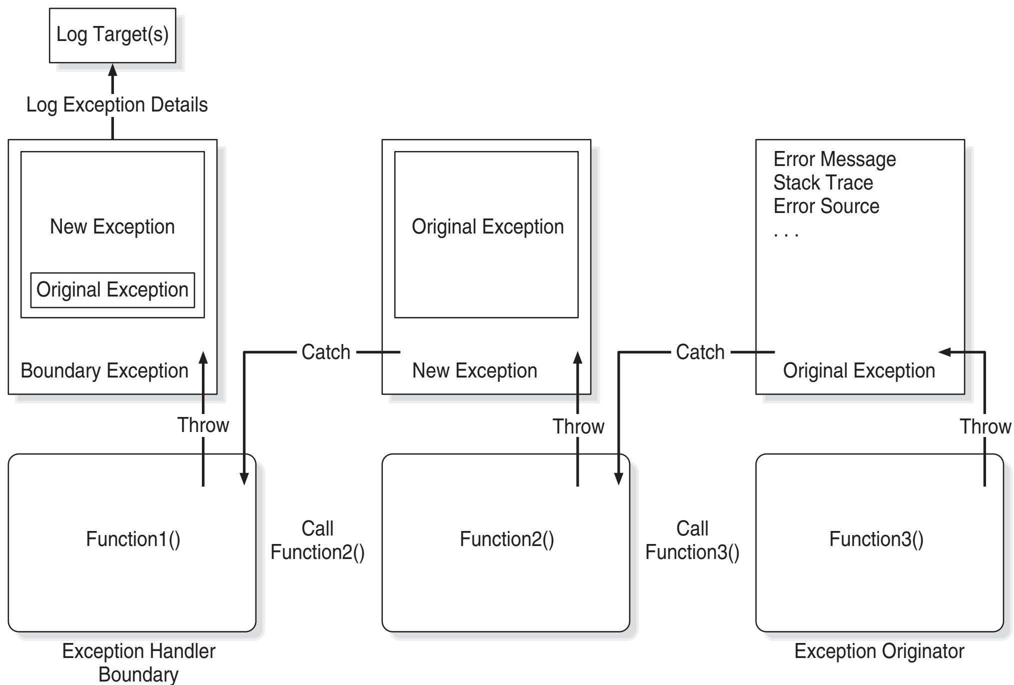


Figure 2.2: Exception chaining: How exceptions are chained using the InnerException property.

- Custom Remote Exception Handling Using SOAP—For exceptions thrown directly from Web services, this can circumvent the FCL's own default SOAP exception handling. Providing custom handling here will give you a greater level of control, especially when formatting an *SOAP Fault*, as we will see in the SOAP Fault technology backgrounder later in this chapter.

- Error Logging and Message Tracing—Provides the mechanism by which the exceptions are communicated and/or made durable during handling.

Figure 2.2 should give you a visual idea of how exception handling, chaining, and logging look from a framework perspective.

Logging is not the most important of these base exception features. Most errors in some way at some point should be made durable. The user may *not* have the luxury of viewing a message box with an error description or even understand the error message when it is displayed. It is up to you to provide a facility for recording

or routing error messages so that the error either gets handled later or is read by those who can help. As a product support feature, this is a must. In the next section, I will show how you can enrich your exception handling by designing a base class to facilitate this.

CHALLENGE 2.1

Besides what has been mentioned, what other elements could be included in the base exception class?

A solution appears in the next section.

BUILDING A BASE EXCEPTION CLASS

Recording an error message does not necessarily have to be fast. I am not saying you should ignore sound design practices and provide a sloppy means of error logging. What I am saying is that your exception handling should concentrate on providing the *best* options for recording errors and not necessarily the *fastest* means of handling them. When an application fails and it begins to throw exceptions, its performance takes a back seat to that of dealing with the error itself. This is not always the case, because there are systems that require fast failing. However, the majority of the systems I have built required that the framework allow the best means of solving the problem. The first and best option for solving the problem is discovering what the problem is. Unless you are the user of the application, this may not be possible without a log and/or tracing mechanism.

Logging allows anyone to read the state of the system at any time. One of the best places to log is in the exception class itself because the exception has the necessary information to record. Logging can be done in one shot, usually in the final catch block of what could be multiple levels of exception handlers. Another option is to log in real time. Real-time logging is the recording of information while the system is running. This is also referred to as *tracing*. This helps eliminate the more difficult runtime problems that may occur and that one-time logging may not help to solve. The trick is to capture as much information as possible in the one-time log event so that tracing isn't always necessary in production. Why? Aside from slowing down performance (which you can minimize, as we will see), tracing forces the developer periodically to add tracing function throughout the code, thus cluttering it up. Before we go into how you can architect a robust and flexible

tracing system, let's first show how we can build one-time logging into our exception handling.

You saw in the *FtpException* example how the user had the option of logging by passing in *true* as the last parameter (*bLog*). However, all of this functionality was passed onto the base class. Because logging usually conforms to a universal standard in your architecture, this can be implemented in your base exception handling class, as shown below:

Listing 2.3: The beginning of a real base exception class.

```
public class BaseException : System.ApplicationException
{

   private int m_nCode;

   public BaseException(){;}

   public BaseException(string sMessage, bool bLog) : this(sMessage,
         null, bLog){;}

   public BaseException(string sMessage, System.Exception
         oInnerException, bool bLog) : this(null, 0, sMessage,
         oInnerException, bLog){;}

   public BaseException(object oSource, int nCode, string sMessage,
         bool bLog) : this(oSource, nCode, sMessage, null,
bLog){;}

   public BaseException(object oSource, int nCode, string sMessage,
         System.Exception oInnerException, bool bLog) :
         base(sMessage, oInnerException)
   {
         if (oSource != null)
                     base.Source = oSource.ToString();
         Code = nCode;

         // need to add logic to check what log destination we
         // should logging to e.g. file, eventlog, database, remote
         // debugger
         if (bLog)
         {
               // trace listeners should already initialized, this
               // is called to be prudent
              Utilities.InitTraceListeners();

              // log it
```

```
            Dump(Format(oSource, nCode, sMessage,
                                    oInnerException));
        }
}

/// <summary>
/// Writes the entire message to all trace listeners including
/// the event log
/// </summary>
/// <param name="oSource"></param>
/// <param name="nCode"></param>
/// <param name="sMessage"></param>
/// <param name="oInnerException"></param>
private void Dump(string sMessage)
{
      // write to all trace listeners
      Trace.WriteLineIf(Config.TraceLevel.TraceError, sMessage);

      // see Utilities.InitTraceListeners()
      // record it to the event log if error tracing is on
      // The EventLog trace listener wasn't added to the
      // collection to prevent further traces
      // from being sent to the event log to avoid filling up the
      // eventlog
      if (Config.TraceLevel.TraceError)
            EventLog.WriteEntry("PMException", sMessage);
}

public static string Format(object oSource, int nCode, string
                  sMessage, System.Exception oInnerException)
{
      StringBuilder sNewMessage = new StringBuilder();
      string sErrorStack = null;

      // get the error stack, if InnerException is null,
      // sErrorStack will be "exception was not chained" and
      // should never be null
      sErrorStack = BuildErrorStack(oInnerException);

      // we want immediate gradification
      Trace.AutoFlush = true;

      sNewMessage.Append("Exception Summary \n")
                  .Append("-------------------------------\n")
            .Append(DateTime.Now.ToShortDateString())
                  .Append(":")
            .Append(DateTime.Now.ToShortTimeString())
                  .Append(" - ")
                  .Append(sMessage)
```

```
                    .Append("\n\n")
                    .Append(sErrorStack);

        return sNewMessage.ToString();
    }

    /// <summary>
    /// Takes a first nested exception object and builds a error
    /// stack from its chained contents
    /// </summary>
    /// <param name="oChainedException"></param>
    /// <returns></returns>
    private static string BuildErrorStack(System.Exception
oChainedException)
    {
        string sErrorStack = null;
        StringBuilder sbErrorStack = new StringBuilder();
        int nErrStackNum = 1;
        System.Exception oInnerException = null;

        if (oChainedException != null)
        {
            sbErrorStack.Append("Error Stack \n")
                        .Append("------------------------\n");

            oInnerException = oChainedException;
            while (oInnerException != null)
            {
                sbErrorStack.Append(nErrStackNum)
                            .Append(") ")
                 .Append(oInnerException.Message)
                            .Append("\n");

                oInnerException =
                    oInnerException.InnerException;

                nErrStackNum++;
            }

            sbErrorStack.Append("\n----------------------\n")
                        .Append("Call Stack\n")
              .Append(oChainedException.StackTrace);

            sErrorStack = sbErrorStack.ToString();
         }
         else
         {
            sErrorStack = "exception was not chained";
         }
```

```
        return sErrorStack;
    }

    public int Code
    {
        get {return m_nCode;}
        set {m_nCode = value;}
    }
```

The logging logic in Listing 2.3 is simple. When throwing an exception, the developer has the option of logging by passing in *true* for the log flag, as shown in this FTP client code snippet:

Listing 2.4: Applying our base exception child class.

```
FtpWebResponse = FtpWebResponse.Create(ControlStreamReader);
switch(FtpWebResponse.Code)
{
    case Constants.POSITIVE_COMPLETION_REPLY_CODE:
    {
        // success  - 200
        // this reply is ok
        break;
    }
    case Constants.SERVICE_NOT_AVAILABLE_CODE:
    case Constants.PERMANENT_NEGATIVE_COMPLETION_REPLY_CODE:
    case Constants.SYNTAX_ERROR_IN_ARGUMENTS_CODE:
    case Constants.NOT_LOGGED_IN_CODE:
    {
        throw new FtpException (this, FtpWebResponse.Code,
            FtpWebResponse.GetOriginalMessage(), false);
    }
    default:
    {
        throw new FtpException (this, FtpWebResponse.Code,
          NotValidReplyCodeMessage(FtpWebResponse.Code, "PORT"),
          true); // log this error!!!!!
    }
```

Here I show a custom FTP exception class passing in the response code, an appropriately returned code description, and the source of the error. In the default section of the switch case statement, the developer wishes to log this exception. In the case statement directly above the default, the *FtpException* (for whatever reason) does not log. This is an imaginary scenario, but it quickly shows how to spec-

ify which exceptions are logged and which ones aren't. The logging itself is handled by the exception, as we have seen. But where does the error actually get logged?

CHALLENGE 2.2

How can you globally control where logging output is sent?

A solution appears in the next section.

Determining Where to Log (Using the Trace Object)

In the BaseException class, logging output is eventually written to one or more log targets using the statement *Trace.WriteLineIf*(). Here I use the tracing capabilities of the FCL to determine where to send the output. You can easily write to the *Windows event log*, a log file, or another target directly but I wanted to leverage what I consider one of the more elegant features of .NET—tracing.

```
// write to all trace listeners
Trace.WriteLineIf(Config.TraceLevel.TraceError, sMessage);
```

The .NET Trace class methods can be used to instrument any debug or release build of your application. Tracing can be used to monitor the health of any system during runtime but it can also act as logging mechanism. Whether you use it for logging, monitoring, or both, tracing can be great way to diagnose problems without affecting or even recompiling your application.

The *Trace.WriteLineIf* method above takes only two arguments. The first is a *boolean* parameter where I use what is called a *TraceSwitch* to determine whether I want to trace a message and at what level. Don't worry about the TraceSwitch just yet if you haven't worked with them; I will explain them in the upcoming technology backgrounder. The second parameter is simply the message I want to trace. Trace.WriteLineIf() is called from within the *Dump* method. The Dump method is, in turn, called from within the overloaded BaseException constructor.

Listing 2.5: Sample Base Exception logging routine.

```
if (bLog)
{
      // trace listeners should already initialized, this
      // is called to be prudent
      Utilities.InitTraceListeners();
```

```
            // log it
            Dump(Format(oSource, nCode, sMessage,
oInnerException));
        }
```

You'll also notice a few other goodies in the BaseException class. In Listing 2.5, you see that the *Dump* method calls the *Format* method. This method is used to prepare the output format of the error message. How you format your messages and what information is included is completely design-specific. Here my formatted message prepares a string that includes three main parts. The first part is the *source* of the error, usually passed in as *this* so that I can access the fully qualified type as well as other elements of the calling object. The second part is the error code that will be ultimately included in the recorded message. The third part is an error summary, which is the description of the error provided by the thrower. This is one set in the exception thrown that will actually be logged. The final parameter is the chained exception, which is usually the one just caught. This will be used to build our error stack because it will contain any previously caught and chained errors. The .NET framework provides the mechanism to chain and even a stack trace to use but it does not provide a formatted error stack. This is your job. Providing a full error stack each time you log is optional but it will help determine the cause of the error. This is in case the high-level error does not provide enough clues as to the cause. In fact, I'm sure you have already found that it typically doesn't.

Determining What to Log

Provide Enough Information

When building your application exception classes, especially your base exception class, you will need to determine how much information to provide. Determining this completely depends on the audience. When providing error information to users, you want to give them enough information to determine that there is a problem without inundating them with terse language or technical details that they may not understand. However, this does not mean you forgo technical information as part of your exception. This only means you do not want to display it by default. For example, when displaying an error dialog, you should show only a high-level version of the problem. Optionally, you can also add a Details button to allow the more technically savvy user get more detailed error information if he or she so chooses. You want to provide the option of information without intimidation.

Providing a high-level error description is typically the job of the last handler of the exception. This is referred to as the *exception boundary*. This could be an ASP.NET page or a GUI catching the final exception and displaying it to the user. Again, this does not in any way mean you forgo providing rich information; in fact, you should side on providing more rather than too little. Just hide the complexity from the user. One of the great things about .NET is that you don't have to provide rich information through extensive custom development. The FCL takes care of providing you with helper objects and methods that will give you as much relevant data as you can handle. The following table shows some of the informational items you can easily access using the FCL:

TABLE 2.1: Environmental information: Some examples of information providers from the FCL

Data	Source
Dates and Times	DateTime.Now
Source of Exception	Exception.Source
Type of Exception	Object.GetType
Exception Message	Exception.Message
Current Method	Reflection.MethodInfo.GetCurrentMethod
Machine Name	Environment.MachineName or Dns.GetHostName
Current IP	Dns.GetHostByName("host").AddressList[0].Address
Call Stack	Exception.StackTrace or Environment.StackTrace
OS Information	Environment.OSVersion
Application Domain	AppDomain.FriendlyName
Current Assembly	Reflection.Assembly.GetExecutingAssembly
Root Error Cause	Exception.GetBaseException
Chained Exception	Exception.InnerException

TABLE 2.1: Environmental information: Some examples of information providers from the FCL (cont.)

Assembly Version	Included in AssemblyName.FullName
Thread ID	AppDomain.GetCurrentThreadId
Thread User	Threading.Thread.CurrentPrincipal

You'll notice in Table 2.1 that a couple of the items are retrieved using .NET *Reflection*. Reflection services allow you *introspect* the public members of almost any class, including the System.Exception class. This can come in very handy when displaying detailed information during logging or display and should be explored. For more information on Reflection services, please refer to the technology backgrounder on Reflection in Chapter 4.

Building an Error Stack

One of the first things I typically provide is an *error stack*. This is not to be confused with a *call stack*; something the System.Exception class provides "out of the box." As mentioned earlier, the error stack is built around another nice feature of .NET called *exception chaining*. This is actually a well-known design pattern that the .NET System.Exception class provides for free. All you have to do is take advantage of it. This means making sure during each throw of your BaseException that you pass in the exception class initially caught, as in the following snippet:

```
catch(FtpException ftpe)
{
      throw new BaseException(this, 1234, "FTP Error: " +
                  ftpe.Message, ftpe, true);
}
```

Here I am throwing a new exception, passing in a *source* object (*this*), an error code (1234), an error description, and what is now the *inner exception*. In this case, that inner exception is an *FtpException*. The last parameter (*true*) will tell this exception to log itself. This will eventually lead to the *BuildErrorStack* method being called. This is the heart of this method. The BuildErrorStack method simply loops through each chained exception, using the inner exception property of the BaseException class, as shown in Listing 2.6. The format and additional items you

place here are, again, up to you. The point is that you leverage the FCL chaining facility by building a useful display medium that will hopefully help solve the issue.

Listing 2.6: Sample stack builder used in our base exception class.

```
if (oChainedException != null)
{
      sbErrorStack.Append("Error Stack \n")
                  .Append("------------------------\n");

      oInnerException = oChainedException;
      while (oInnerException != null)
      {
            sbErrorStack.Append(nErrStackNum)
                        .Append(") ")
                    .Append(oInnerException.Message)
                        .Append("\n");

            oInnerException =
                  oInnerException.InnerException;

            nErrStackNum++;
      }

      sbErrorStack.Append("\n----------------------\n")
                  .Append("Call Stack\n")
          .Append(oChainedException.StackTrace);

      sErrorStack = sbErrorStack.ToString();

}
```

CHALLENGE 2.3

When throwing exceptions from Web services, what other steps are required? How does SOAP affect exception handling?

A solution appears in the upcoming technology backgrounder on SOAP Faults (page 65).

Throwing System Exceptions

Once you've built your base exception class and all of its helpers, you are probably going to want to leverage it as much as possible. That said, you *may* also want to take advantage of some of the *system exception classes* (e.g., System.IOException) that the FCL has to offer. How does one leverage the features of both? The answer

is that you use both by employing *exception wrapping*. When you want to throw a system exception and still want to take advantage of your base exception features, you simply create a new system exception class and pass that as the inner exception of your base exception class constructor. This in effect "wraps" the system exception, allowing it to appear neatly as part of your error stack. This can all be with one (albeit long) line of code:

```
throw new BaseException(this, 120, GetMessage(120),
new System.IOException(), true);
```

Here I want to throw the system exception IOException. However, I still want to use my BaseException class features, such as error stacks, message formatting, and logging. Using exception wrapping, I get both.

MANAGING EXCEPTION BOUNDARIES

Once I've added all of my robust error handling-features to my exception classes, I am now positioned to easily determine faults, should they arise. However, there is more than just providing the error-handling mechanisms. You must also provide a model to follow for how exceptions will be handled by answering some of the questions below:

1. Where is the exception handled?
2. When should you log an error?
3. Will the exception be "remoted?"
4. Who will be the final receiver of the exception?
5. What protocols and transports is the receiver using?

This section discusses the answers to these questions and will provide you with a starting point upon which to build an *exception boundary* policy.

Throwing Exceptions from Web Services

In most Web applications, the last boundary of properly-handling exceptions is controlled by the *ASP.NET* or *Web service* code. In most cases, throwing exceptions from a Web service can be done in the same manner as throwing exceptions from

anywhere else. You simply throw an exception from the Web method, and .NET takes care of the rest. If SOAP is used to call the Web method, .NET automatically wraps your exception in a corresponding *SoapException* class, nesting your original exception within it. However, in many cases the details of that wrapped exception may not suit your client application. For example, wrapping the exception in this manner may not provide the level of detail your clients require. This is especially true when the message protocol used is SOAP. Thankfully, there is a way to circumvent this automated wrapping by throwing your own SOAP exceptions in cases where custom-detailed information may be required. Figure 2.3 should give you a good visual of how exceptions are wrapped inside of a SoapException class. Notice there is no loss in the amount of detail because all information is preserved through chaining.

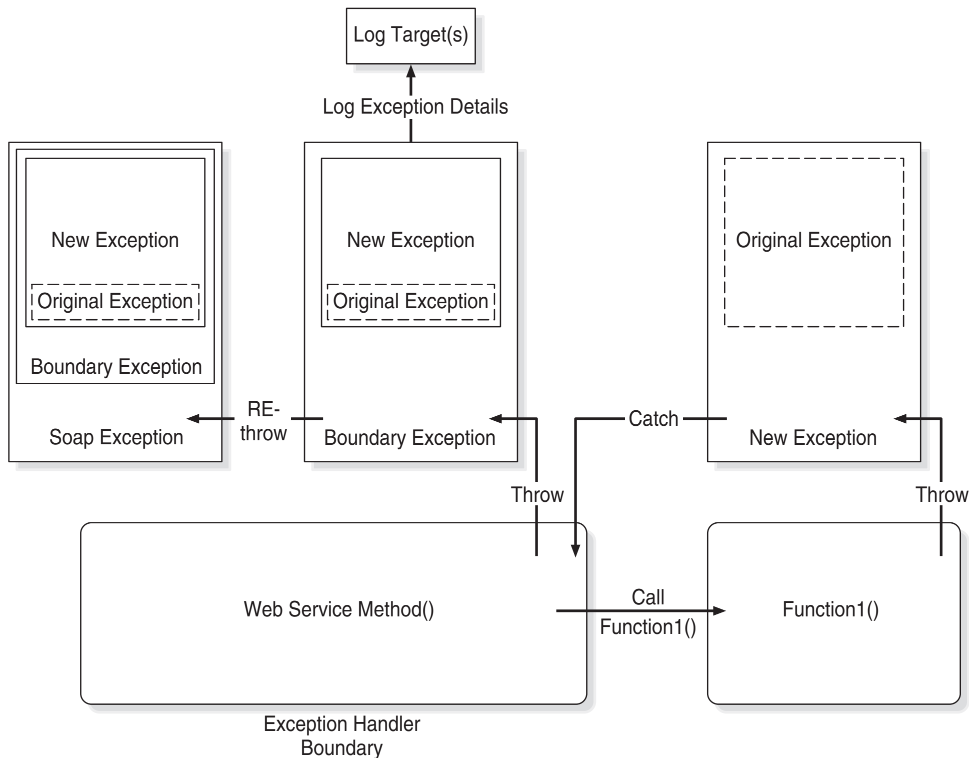


Figure 2.3: Wrapping SOAP exceptions: How base exceptions are wrapped by SoapExeptions.

This should by no means discourage the client from using SOAP—quite the contrary. Unless messages passed to Web services are rather simple and, thus, using a HTTP GET or POST is sufficient, I suggest using SOAP as your message protocol for passing data. Using SOAP will provide you with the most flexibility in passing complex types, such as *DataSet*, to your prospective Web services. With .NET, you do not have to even determine which message protocol is used. Using this wrapping technique, .NET detects the message protocol and will wrap your thrown exception in a SoapException class. However, the details provided are sparse, at best. This is especially true with SOAP Fault information. In these cases, the best practice is to throw your own SoapException. For those SOAP 2.0 clients, this means they will receive a properly formatted SOAP Fault on the client with as much detail as required. A SOAP Fault is a mechanism used to determine the details of the exception in a "SOAP-compliant" manner, such as the following technology backgrounder describes. A SOAP Fault is not simply used for low-level errors but can be the means by which all exceptions are determined, as we will see in the upcoming implementation pattern.

TECHNOLOGY BACKGROUNDER—SOAP FAULTS

As with all SOAP messages, the content of the message is included in the body and is processed on the server. In this case, the server is a Web service method. The body element contains the information necessary to invoke a Web service and will also be used to return any values upon successful completion of the business function. However, if any error were to occur, the body can also be used to return errant information in the form of SOAP Faults. The SOAP Fault element is used to carry this error and/or status information back to the calling client from within the SOAP message. If an error occurs, the SOAP Fault element should appear just like any other body element entry and should not appear more than once.

Remember that the SOAP message can be parsed like any other XML document. To determine whether an error occurs, the calling client can simply look for this fault element and read its contents to determine the exact error. Fortunately, the client from the SOAP Toolkit 2.0 provides users with a user-friendly component by which to query for this information. The SOAP client COM object simply wraps the nested contents of the fault, allowing access to the following elements:

faultcode

This element is used to determine the basic error identification and to provide an algorithmic mechanism for identifying the fault. This must be present in a SOAP Fault element, and it must be a qualified name and not the typical numbering schema used in other error determinations. The namespace identifier for these *faultcode* values is http://schemas.xmlsoap.org/soap/envelope/. The following *SOAP* faultcodes can be used:

VersionMismatch—Used for an invalid namespace in the *SOAP Envelope* element.

MustUnderstand—An element of the *SOAP Header* that was missing or invalid. See the SOAP Header technology backgrounder in Chapter 7 for more information.

Client—Used to indicate that the message was incorrectly passed and/or did not contain the right information used for the called function. This simply means it was the client's "fault" and should be retried. Retrying or repassing the information will not result in the proper processing on the server unless the contents of the passed message are changed. This is not the server's error. You set this when the caller gives the service improper information and not when something goes wrong on the server side.

Server—This is used to communicate a server failure (this can mean any failure to process the request from the server's perspective). Retrying or repassing the information may result in the successful completion of the function.

faultstring

This element is used to pass to the caller a descriptive human-readable error. It must be present in a SOAP Fault element and should provide at least some information explaining the nature of the fault. This could contain the high-level error message used to determine generally what went wrong.

faultactor

This element is used to provide information about who caused the fault and usually contains the Uniform Resource Identifier (URI) of the perpetrator. This is similar to the SOAP actor attribute in the SOAP body. Applications that do not act as the ultimate destination of the SOAP message must include this element. The final destination of a message may use this element to indicate that it alone threw the exception.

detail

This is where the most descriptive error information can be included to help determine the root cause of the problem. The detail element can be used for carrying application-specific error information related to the Web service method invoked. It should not contain SOAP header-related errors but only errors related to the message body. The absence of this element shows that the fault is not related to the Body element. This can be used to distinguish whether the Body element was even processed and is your key to show that the message was received but that an application-specific error was indeed thrown. This element is also the key to the implementation pattern: *SOAP Exception Fault Builder*. According to the SOAP specification, "other *SOAP* Fault subelements may be present, provided they are namespace-qualified."

An example of a SOAP Fault generated from the Microsoft SOAP Toolkit 2.0 *SoapServer* object is as follows (here it places an <errorInfo> element in the <detail> element to convey detailed error information to the client):

Listing 2.7: Sample Soap Fault Detail Block.

```
<soap:Fault …>
 …
<detail>
   <mserror:errorInfo
         xmlns:mserror="http://schemas.microsoft.com/soap-
           toolkit/faultdetail/error/">
   <mserror:returnCode>-2147024809</mserror:returnCode>
   <mserror:serverErrorInfo>
         <mserror:description>Failure...shutting down</
mserror:description>
         <mserror:source>error-source</mserror:source>
         <mserror:helpFile>
               help goes here
         </mserror:helpFile>
         <mserror:helpContext>-1</mserror:helpContext>
   </mserror:serverErrorInfo>
   <mserror:callStack>
         <mserror:callElement>
              <mserror:component>SomeOp</mserror:component>
              <mserror:description>
                    Executing method ReturnError failed
              </mserror:description>
              <mserror:returnCode>-2147352567</mserror:returnCode>
        </mserror:callElement>
  </mserror:callStack>
 </mserror:errorInfo>
</detail>
```

This contains all of the application-specific error information as generated from the Toolkit's SoapServer object. Your own custom detail element does not necessarily need to look like this but this provides a decent example of what can be done. The more information you provide, the more robust your error handling will become, especially when this information is provided in a SOAP Fault-friendly manner.

As mentioned earlier, throwing normal exceptions using the .NET framework's SOAP exception wrapper may not always provide the details you were expecting. In fact, the main error message provides only the same generic text message: "Server was unable to process request." To provide a slightly more robust SOAP exception with a custom SOAP Fault, you need to throw your own SoapException class. By doing, so you can provide a much richer exception and still provide a SOAP-compliant fault mechanism so that all clients can better determine the error generated by the Web service. Fortunately, there is a simple way to throw SoapExceptions from your Web service without requiring much effort. In the next section, I will show you exactly how to implement this pattern by adding support to your base exception class. See the beginning of this chapter for details on building a base exception class.

Throwing Custom SOAP Exceptions

In some cases, it is difficult to find areas where Microsoft has not provided a feature desired in its framework. In fact, you have look a little closer before realizing where you could truly add benefit by adding custom behavior. Such is the case for processing exceptions from Web services when using SOAP.

As mentioned in the previous section, when you throw an exception from a Web service that is called from SOAP .NET, this automatically wraps your exception class in the *SoapException* class. For a remote client, receiving that exception will come in the form of a partially populated SOAP fault. I say *partially* because .NET does not fill in every detail. I've built a simple Web service to demonstrate this, as shown in Figure 2.4.

In Figure 2.4, you'll find a Web service called *ExceptionThrower.asmx* displayed in a browser. Here you simply specify whether you want to throw a custom SOAP exception using the following implementation pattern or using the framework wrapper. Specifying True in the edit box and selecting Invoke, you should see the output shown in Figure 2.5.

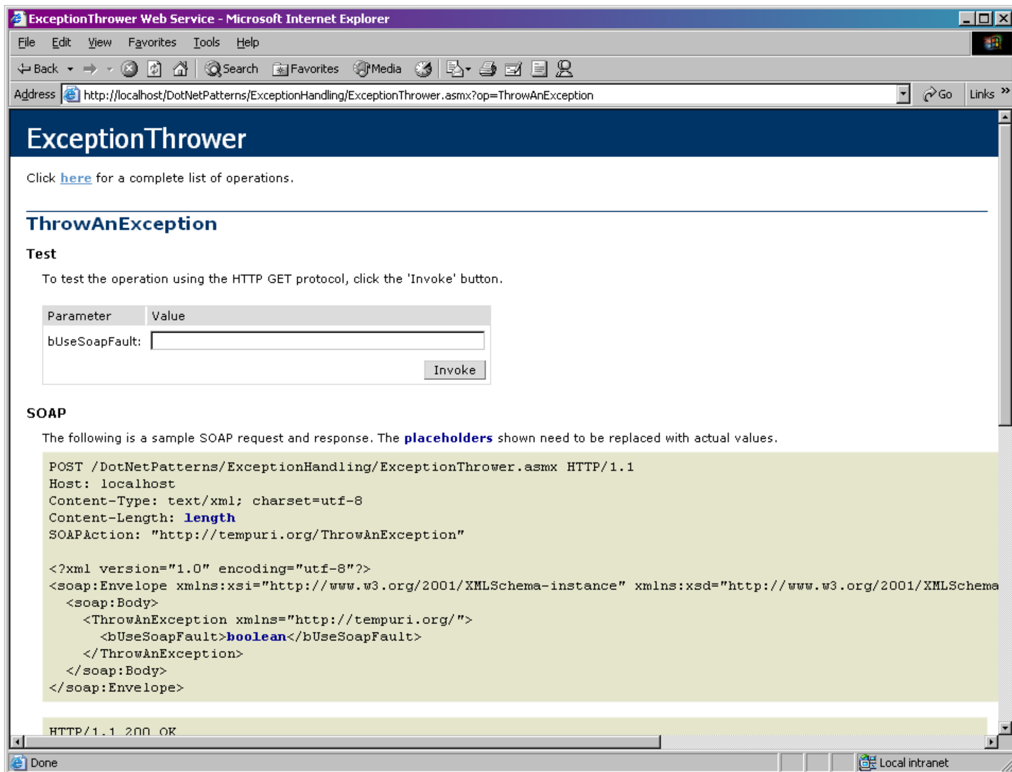


Figure 2.4: ExceptionThrower.asmx test harnesses a Web service for throwing custom SOAP exceptions.

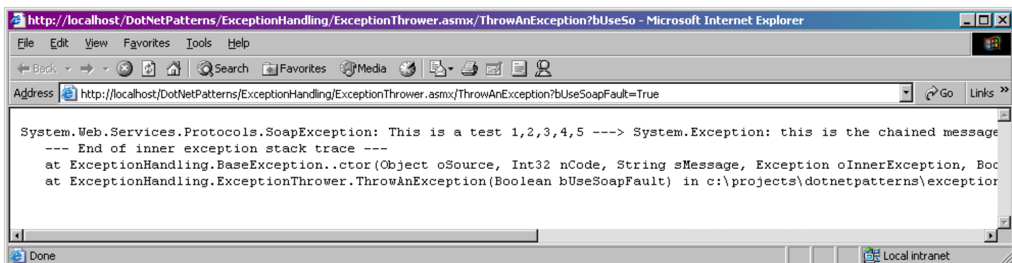


Figure 2.5: ExceptionThrower.asmx exception output when selecting custom soap exception.

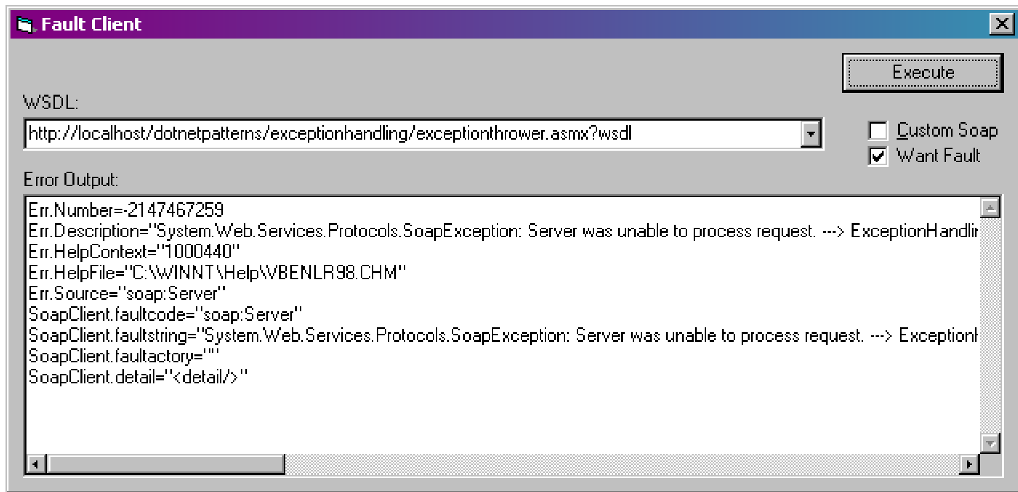


Figure 2.6: Called from a SOAP 2.0 client written in VB 6.0 (using wrapper SOAP exceptions). Does not show any SOAP Fault details.

If you run this again but select False, you should *not* be able to view the exception text. This is an apparent side effect to *not* throwing your own SOAP exceptions, at least for Beta 2. This, however, is not the real reason to throw your SOAP exceptions. To the see the real benefit of throwing your own custom SOAP exceptions, run the exceptionthrower.asmx Web service from a typical SOAP 2.0 client (Figure 2.6).

Using the test GUI, I am *not* selecting Custom Soap before I select Execute. This will call the same Web service with a parameter of False, thus using the .NET wrapper method. Notice that .NET wraps our exception in a SoapException but it provides us with a general error text before the actual error message. More critically, it does not provide us with any details (the <detail/> element is empty). Finally, it does *not* tell us where the error came from in the form of the *faultfactory* property. Now when we turn on the Custom Soap option, we should receive a much more detailed SOAP Fault (Figure 2.7).

Notice in the previous screen shot that a few things have now been provided. The faultcode property has been set to *detail*. This is optional but now we have the flexibility to set this faultcode appropriately. For more information on SOAP Fault codes, please refer to the relevant technology backgrounder in this chapter. Also notice that the general description has been replaced with our specific error message. Next, the faultfactory property has been set to the location of the exception. In this case, that location is the actual URI of the Web service. Finally and most

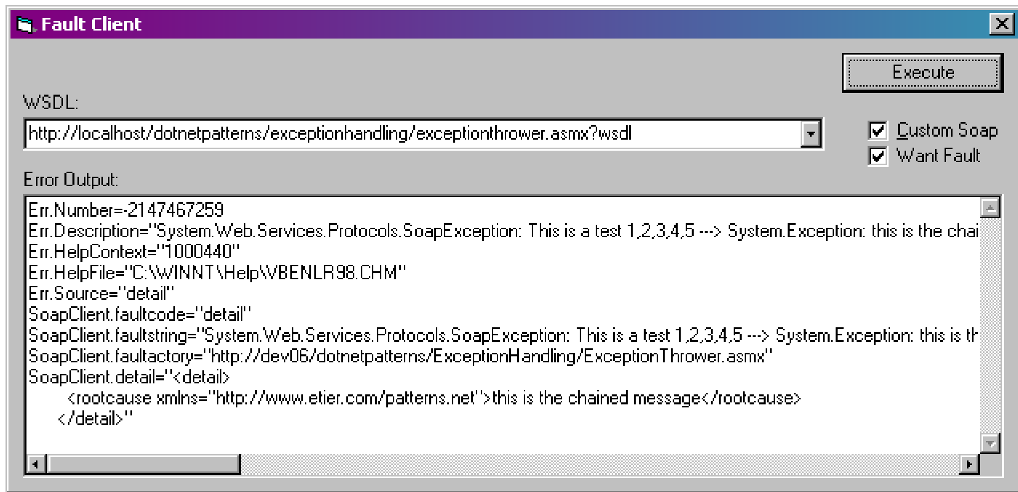


Figure 2.7: Called from a SOAP 2.0 client written in VB 6.0 (using wrapper SOAP exceptions). Shows the SOAP Fault details.

important, the *details* property has been set to the *root cause* of the error. The information provided in this property can be as detailed as you like. In fact, this would be a great place to display an error stack; as long as the details are properly formatted as SOAP-compliant, the choice is yours. Next I will show you the code behind this implementation pattern.

Adding SOAP Exception Support to BaseException

In this implementation pattern, I will place the custom SOAP exception inside of our BaseException class. This provides two benefits. First it encapsulates the details of the SoapException class from the thrower of the exception, such as a Web service method. Second, it simplifies the mechanism by which all exceptions are thrown. The easier and more standard you can make exception handling, the better. The exceptionthrower.asmx Web service, shown below, shows how to throw or *not* to throw a custom SOAP exception:

Listing 2.8: Sample Soap Fault Builder for our base exception class.

```
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
```

```
namespace ExceptionHandling
{
   . . .

   [WebMethod]
   public void ThrowAnException(bool bUseSoapFault)
   {
         if (bUseSoapFault)
               throw new BaseException(this, 1234,
                           "This is a test 1,2,3,4,5",
                           new Exception(
                           "this is the chained message",
                           null), false, true);
         else
               throw new BaseException(this, 0,
                           "This is a test 1,2,3,4,5",
                           new Exception(
                           "this is the chained message",
                           null), false);
         }
   }
}
```

The code in Listing 2.8 simply passes *true* or *false* as the last parameter of one of our BaseException class constructors. If you pass *true*, the first BaseException shown is thrown; otherwise, the same signature we used in the constructor defined in the beginning of this chapter is used.

Passing *true* will call the following constructor.

Listing 2.9: Sample BaseException ctor for allowing Soap Faults.

```
/// <summary>
/// This ctor for throwing soap exception usually from web service
/// methods
/// This will format error message in more soap fault friendly
/// manner
/// filling in fields not filled in by the default wrapper exception
/// method in .NET
/// </summary>
/// <param name="oSource"></param>
/// <param name="nCode"></param>
/// <param name="sMessage"></param>
/// <param name="oInnerException"></param>
/// <param name="bLog"></param>
/// <param name="bThrowSoap"></param>
public BaseException(object oSource,
```

```
                    int nCode,
                    string sMessage,
                    System.Exception oInnerException,
                    bool bLog,
                    bool bThrowSoap) : this(oSource,
                                            nCode,
                                           sMessage, oInnerException,
bLog)
{
   string sCause = "no root cause found";
   XmlNode oNode = null;

   if (bThrowSoap)
   {
         if (oInnerException != null)
               sCause = oInnerException.GetBaseException().Message;

         // now build the details node to use for the soap exception
         // -- use the root cause for the main text oNode =
            BuildDetailNode(sCause);

         Trace.WriteLine("Throwing Custom Soap Exception - Message "
                           + sMessage);

         // build actor or source uri to set into actor field -
         // replace .'s with /'s to make a uri from it
         // NOTE: the web service must namespace match the reqeues
         // uri if this is to be accurate
         // we can't use context without passing it so we'll build
         // one
         StringBuilder sActor = new StringBuilder("http://");
         sActor.Append(Dns.GetHostName());
         sActor.Append("/");
         sActor.Append("dotnetpatterns");
         sActor.Append("/");
         sActor.Append(oSource.ToString().Replace(".", "/"));
         sActor.Append(".asmx");

         throw new SoapException(sMessage,
                           SoapException.DetailElementName,
                                 sActor.ToString(), oNode,
                                 oInnerException);
   }
}
```

There are two main parts to this pattern. First, there is the building of the details property so that we can fully extract SOAP Fault details from the client. The second is the fact that I am “rethrowing” a *SoapException* back, once the properties

are filled. To rethrow a SoapException, we simply call *throw new SoapException* like any other exception, and the exception should be thrown back to the calling client. Before I do that, however, I need to fill four properties that will become the SOAP Fault elements.

Each of the four properties that make up the SOAP fault will be passed to the SoapException constructor, as shown in Listing 2.9. The first parameter is any message text. Typically, as is the case here, it will be the high-level error message text we've sent when throwing our main exception. The second parameter designates the SOAP Fault code, as defined by the SOAP specification. Although this can be set to any code and your code will still work, it is considered "well formed" if you comply with standards. For this example, I've set this parameter to the detail property because I will be providing a detail explanation as part of the fourth parameter. The third parameter is the *actor* and is used to set the location of where the error occurred in the form of a URI. Here I dynamically determine the full URI path based on the *source* object provided and the machine on which this service is running. The oSource.ToString() will give me a fully qualified path string, including the namespace of the assembly under which the source object resides. This will only work if your Web service path matches this fully qualified name and is used only as an example here.

Finally, the fourth parameter provides the details section of the SOAP Fault and is a little more complicated to build so I've supplied a helper method called *BuildDetailNode*, as shown here:

Listing 2.10: Sample Soap Fault Detail Node Builder.

```
/// <summary>
/// Build the xml node used for throwing custom soap exceptions, the //
/ root cause string will be use for the main content
/// </summary>
/// <param name="sCause"></param>
/// <returns></returns>
public System.Xml.XmlNode BuildDetailNode(string sCause)
{
   XmlDocument oDoc = new XmlDocument();
   XmlNode oNode = oDoc.CreateNode(XmlNodeType.Element,
                     SoapException.DetailElementName.Name,
                SoapException.DetailElementName.Namespace);

   // Build specific details for the SoapException.
   // Add first child of detail XML element.
```

```
   XmlNode oDetailsNode = oDoc.CreateNode(XmlNodeType.Element,
               "rootcause", "http://www.etier.com/patterns.net");
   oDetailsNode.InnerXml = sCause;

   oNode.AppendChild(oDetailsNode);
   return oNode;
}
```

This method uses the *System.Xml.XmlDocument* and *XmlNode* classes to build a SOAP Fault-compliant XML node. Once built, this node will be passed back and used as our detail SOAP Fault property. This can be built any way you see fit, as long as the parent detail element node exists along with its namespace; otherwise, it will not be properly displayed in your SOAP client. This SoapException.DetailElementName type is used from within our first CreateNode() call for just this purpose. After this step, you can supply your own text or custom-nested elements. In this example, I add another child node called *rootcause* and specify the text of the root cause or error, as supplied by the System.Exception class. Another variation of this pattern would be to add the full error stack as child nodes so that the SOAP fault client can see every detail of the error. Either way, the choice is yours, so have fun with it.

COM (Interop) Exception Handling

The common language runtime (CLR) seamlessly provides the ability to access COM objects, as well as allowing COM clients to access .NET code. The CLR's interoperability features also provide sophisticated exception handling when the managed layer is crossed during error scenarios. When a COM client calls a managed piece of code that throws an exception, the CLR intercepts the exception and appropriately translates it into an HRESULT. Similarly, if managed code is accessing an unmanaged COM component that returns an HRESULT, the CLR translates it into an exception. While doing so, the CLR incorporates any additional information provided by the COM *IErrorInfo* interface into the thrown managed exception class. Any unrecognized HRESULT will result in a generic *ComException* being thrown with the *ErrorCode* set to the unresolvable HRESULT.

When using COM components from managed code, this does not present many problems. However, when COM clients currently access your managed code, many of the FCL exceptions will return an HRESULT that may be unrecognizable to the calling COM client. Cases where the calling COM client must handle certain HRESULTs in an application-specific manner can cause problems.

Fortunately, the System.Exception object has an *HResult* property that can be overridden and set at any time during your error handling. To avoid problems with legacy COM clients, replace any HRESULT with that of one that is recognized by those clients to avoid such problems.

Using XML

The output format is completely up to you. A proprietary format was used in the previous examples but XML could have been used. Be forewarned, however, that this output may have multiple targets, each with its own viewers. For example, XML may not be the best display format if you are writing to the Windows 2000 Event Log. A large XML stack does not look particularly great in the Windows Event Viewer.

Another point to keep in mind is when building XML documents, be careful how you format any error strings. If the error strings contain characters interpreted as XML syntax, you may have problems formatting such errors. For example, in the previous section we built a detail XML node that made up one of the properties of our SOAP Fault. The text that was used in the detail node was the root cause of the error. If any of the text strings used for the message inside of the detail XML node happened to contain a "<" or ">" character, a format exception would have been thrown. Having errors occur in your own error handling is not something you want to have happen because it may make communicating them impossible. In fact, you may be a little embarrassed that your own error-handling code has bugs. This is not to say that you should avoid XML in your error handling. Rather, it is quite the contrary. I point this out only so that you can be extra careful when building your base exception class. The more you test your code in this case, the better.

Determining When to Log

Determining when to actually log your output can easily become one of those heated design debates. To keep things simple, however, I suggest logging at the most externally visible tier, or what I've been calling the *exception boundary*. This coincides with what .NET considers an application and where you specify your configuration files. This is also known as the *application level*. For GUI development, this is simple: Log just before you display your error. For Web service applications or for those that do not necessarily have a visible tier, I typically log just before returning from an external Web method. This refers only to error logging. Random tracing is another story and is completely dependent on how much out-

put you want to receive and when you want to begin seeing it. For those really nasty bugs, providing a real-time, persistent tracing scheme where you trace as much as possible may be your best option. There is a way to have the best of both worlds—the ability to trace verbose information but only during times where it may be deemed necessary. We will be covering this practice in the next section.

ENABLING THE TRACE OPTION

By default, *C#* enables tracing with the help of Visual Studio.NET by adding the /d:TRACE flag to the compiler command line when you build in this environment. Building under Visual Studio.NET automatically will provide this switch for both debug and release builds. Therefore, any tracing can be viewed in either release or debug, which is something I am counting on for the following implementation pattern. Another option for adding tracing is to add *#define TRACE* to the top of your *C#* source file. The syntax and mechanism to enable tracing is compiler-specific. If you are working with VB.NET or C++, for example, your settings are slightly different. However, each .NET language should support tracing, so refer to your documentation for details.

CHALLENGE 2.4

How would you send logging output (tracing output) to a remote terminal across the Internet?

A solution appears in the upcoming section on remote tracing (page 82).

The target of the trace output is then determined by what are called *trace listeners*—the subject of the next technology backgrounder. If you are currently already familiar with trace listeners and dynamic tracing using trace switches, you can skip this section.

TECHNOLOGY BACKGROUNDER—TRACE SWITCHES AND TRACE LISTENERS

Trace Listeners

Trace listeners are thread-safe classes that derive from an abstract class called, appropriately enough, *System.Diagnostics.TraceListener*. This class contains the

necessary methods to be able to control trace output. There are three out-of-the-box trace listeners included with .NET. The first is called *DefaultTraceListener* and, by default, is automatically added to a trace listener collection shared by the application. Using the *Trace.Listeners* static property, you can access this collection at any time during execution. Using the collection, you can then add or remove any trace listener. Controlling the listeners in the collection will, therefore, control your trace output. When *Trace.Write* or *Trace.WriteLine* is called, these methods will emit the message to whatever trace listeners have been added to the system. The DefaultTraceListener will receive these calls and send the output to the its designated target. For the DefaultTraceListener, that target is both the *OutputDebugString API* and the .NET log method. You should already be familiar with the OutputDebugString API; if not, please reference the Platform SDK documentation. The log method will post the message to any attached debugger in .NET.

The second trace listener is called *EventLogTraceListener*. This listeners send all output to the Windows event log. Using its *EventLog* property, you can control which event log receives the output. The third out-of-the-box listener is called *TextWriterListener*. This sends all output to any *TextWriter* or *Stream* object. This can be the *console's standard output stream* or any file.

To add or remove a listener to or from the collection, you can either use a configuration file, such as *web.config*, or do it in code. The following is an example of adding or removing a listener using a configuration file:

Listing 2.11: Adding a listener using web.config.

```
<configuration>
  <system.diagnostics>
    . . .
    <trace autoflush="true" indentsize="4">
      <listeners>
        <add name="LogFileListener"
   type="System.Diagnostics.TextWriterTraceListener,System"
          initializeData="c:\LogFileListener.log" />
          <remove
    type="System.Diagnostics.DefaultTraceListener,System"/>
      </listeners>
    </trace>
  </system.diagnostics>
</configuration>
```

You can also control the listener collection in code such as the following *InitTraceListeners* method I use in the upcoming implementation pattern.

IMPORTANT

In the following method, I show how you can add an event log listener to the collection. However, I do not recommend you use the event log during most tracing scenarios. I typically use the event log only during error handling or more determined forms of tracing, such as message coming from Windows Services (e.g., "service starting…", "service stopping…", etc.). Otherwise, you will quickly fill it up if you're not careful.

Listing 2.12: Sample for adding an event log listener to a global collection.

```
/// <summary>
/// Adds all default trace listeners, for event log
/// tracing it first checks
/// to see if the trace level has not been set to verbose or
/// information since we
/// don't want to fill up the event viewer with verbose
/// information.
/// </summary>
public static void InitTraceListeners()
{
      FileStream oTextWriter = null;

      // We do not want to dump to the if tracing is set to
      // information or verbose
      if (!Config.TraceLevel.TraceInfo)
      {
            if (Trace.Listeners[TRACE_EVENTLOG_KEY] == null)
            {
                  EventLogTraceListener oEvtLogListener = new
                    EventLogTraceListener(EVENTLOG_SOURCE);
                  oEvtLogListener.Name = TRACE_EVENTLOG_KEY;
                  Trace.Listeners.Add(oEvtLogListener);
            }
      }
      else // travel level is set to warning or error
      {
            if (Trace.Listeners[TRACE_EVENTLOG_KEY] != null)
                  Trace Listeners.Remove(TRACE_EVENTLOG_KEY);
      }
```

```
    if (Trace.Listeners[TRACE_TEXTWRITER_KEY] == null)
    {
          oTextWriter = File.Exists(TRACE_LOG_FILE)) ?
                        File.OpenWrite(TRACE_LOG_FILE) :
                        File.Create(TRACE_LOG_FILE);
          Trace.Listeners.Add(new
                  TextWriterTraceListener(oTextWriter,
                            TRACE_TEXTWRITER_KEY));
    }

    // This is a custom trace listener (see PMRemoteTrace.cs)
    // for remote tracing
    if (Trace.Listeners[TRACE_REMOTE_KEY] == null)
          Trace.Listeners.Add(new
                      RemoteTrace(TRACE_REMOTE_KEY));
}
```

First, notice that the listener collection is manipulated like any other collection in .NET. Using .NET *indexers*, I can check to see whether the listener has already been added to the collection. If you look carefully at the *InitTraceListeners* example, you'll notice a new listener class called *RemoteTrace*. This is referred to as a *custom trace listener* and is the focus of the following implementation pattern.

By adding trace listeners to a central collection, you can globally control how and where tracing output is sent. As a developer of the rest of the system, you are required to add Trace.Write or WriteLine methods only to send the appropriate information to that output for any debug or release build. This is a powerful monitoring and logging feature but how do we elegantly control whether we want to trace at all? For that matter, how to we control at what level we would like to trace? We may simply want to trace errors or we may want to provide as much information during runtime execution as possible to help determine our problems. This is where .NET *switches* come into play.

Boolean and Trace Switches

The first way to turn off any tracing is by disabling the /d:TRACE option, as mentioned above. However, doing so requires the recompilation of your code for it to become effective. What if you are in production and you do not have that option? Fortunately, you can easily control tracing dynamically and without recompilation. To do so, you use what are called *System.Diagnostics.Switch* objects. The switch objects available in the FCL are *BooleanSwitch* and *TraceSwitch*. Both derive from System.Diagnostics.Switch. All switches are configuration objects, of sorts,

that read a configuration setting and provide properties with which you can check dynamically to determine whether an option has been enabled and what level. The *BooleanSwitch* is the simpler of the two and is set to off by default. To turn it on, you edit your application's *<xxx>.config* file as follows:

```
<system.diagnostics>
      <switches>
            <add name="MyBooleanSwitch" value="1" />
      </switches>
</system.diagnostics>
```

The .config file can be your web.config file if your application is a Web service, for example. With this set, you now create the BooleanSwitch object. This can be stored in a *static* variable and be accessible by any code wishing to use the switch. This BooleanSwitch data member is part of a configuration object used throughout the system to retrieve any application configuration information.

```
public static BooleanSwitch MyBooleanSwitch = new
TraceSwitch("MyBooleanSwitch",
"This is my boolean switch");
```

The first parameter of the BooleanSwitch is the name of the switch. This *must* match the name used in your configuration setting. Setting this to 1 turns on the switch; conversely, setting this to zero turns it off. Once the switch is created, the BooleanSwitch can be used to help determined things such as tracing. Instead of calling Trace.Write or Trace.WriteLine, you now call Trace.WriteLineIf. The first parameter of this method is a Boolean that when set to *true* will cause tracing to occur. Instead of simply passing *true* or *false* directly, you use the BooleanSwitch, as follows:

```
Trace.WriteLineIf(MyBooleanSwitch.Enabled, “message to trace”);
```

During execution, the code will use the BooleanSwitch to dynamically check the configuration setting. If the setting is set to 1, the BooleanSwitch returns *true*, and the trace occurs. No longer do you need to recompile your code.

A TraceSwitch works in the same fashion except you also get to control the level at which tracing should be enabled. To set a TraceSwitch in the configuration file, you must specify the name of the switch (same as before) and the level at which to

enable tracing. A TraceSwitch has four properties: *TraceError, TraceWarning, TraceInfo*, and *TraceVerbose*. Its configuration-level settings are:

```
0 (off), 1 (error), 2 (warning), 3 (info), OR 4 (verbose)
```

For example, to create a TraceSwitch to control all system tracing, use the following:

```
public static TraceSwitch TraceLevel = new
TraceSwitch("TraceLevel", "Tracing Level");
```

To configure this TraceSwitch to verbose, you would use the following:

```
<system.diagnostics>
      <switches>
            <add name="TraceLevel" value="4" />
      </switches>
</system.diagnostics>
```

To trace a message (only if verbose has been set), you would use the following:

```
Trace.WriteLineIf(TraceLevel.TraceVerbose, “some message”);
```

This line will send a trace message only if the *verbose level* has been set in the configuration file. In fact, using any other trace level in the *WriteIf* call would trace the message because setting the configuration to verbose will send all traces. Setting this to verbose causes the TraceSwitch *trace level* to return *true* for TraceInfo, TraceWarning, or TraceError. If the configuration setting was 3 (TraceInfo), then all trace levels except TraceVerbose would return *true* and, thus, the trace would be sent. In this case, TraceInfo, TraceWarning, and TraceError would return *true* but TraceVerbose would return *false*. I will discuss the use of tracing, custom trace listeners, and trace switches in the following section.

REMOTE TRACING—BUILDING A CUSTOM TRACE LISTENER

One of the nice features of using *trace listeners* and *trace switches* is that you can centrally and dynamically control all tracing output from a central location. Taking this a step further, you now have a relatively simple means of improving product support for your applications. Such a step allows you to create your own trace listener class to completely customize the output, as well as the trace target. For

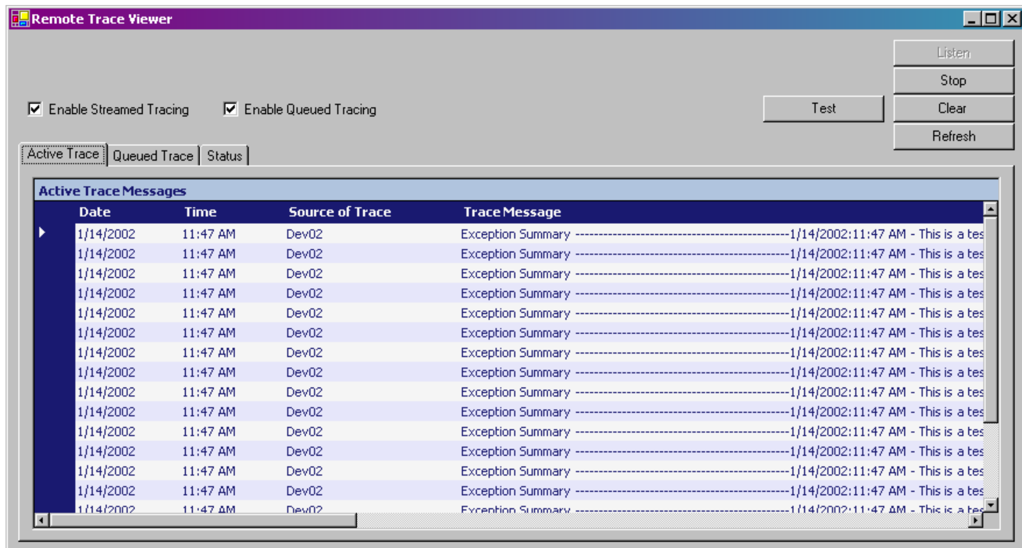


Figure 2.8: Remote Trace Viewer: Displays all remote traces coming from a message queue or TCP/IP socket.

example, you may want to remotely monitor the health of an application that was installed externally. The application you wish to monitor may not even be installed on your local network. With a custom trace listener, you now have the means of automatically tracing output from an external application to a server accessible to you. I will go through the building of such a feature using three main components:

- Custom Trace Listener/Sender—Trace listener to send trace statements remotely.
- Trace Receiver—Server that can receive trace statements from a remote source.
- Trace Viewer (optional)—Front end to view remote traces (Figure 2.8).

Building a Custom Trace Listener

Using the .NET trace listener collection, you can easily employ your own custom listeners to route and format trace output as you see fit. Custom trace listeners are manipulated like any other trace listeners. To create one, you must first derive from the *TraceListener abstract* class and create your own XXX*listener* class. The name of the class can be anything but typically it will end with the word *listener* to

keep with the .NET convention. Once created, you can than add this listener to the listener collection, as we did in the previous section. You are required to implement only two methods to guarantee that your custom listener will work with .NET tracing. At a minimum, you must implement both the *Write* and *WriteLine* abstract methods. You are not required to overload them; one implementation of each is sufficient. Other elements of the TraceListener parent class may also be overridden, such as the Flush or Close methods, but only the Write and WriteLine methods are actually required for compilation. To create our custom remote trace listener, specify something like the code snippet in Listing 2.13.

Please refer to the previous technology backgrounder in this chapter for details on trace listeners and switches.

Listing 2.13: Our Remote Trace Listener template.

```
/// <summary>
/// This represents the remote tracing custom trace listener that ///
will be added during application initialization
/// and allow all tracing to be sent remotely, the tracing level is ///
determined by the trace switch which is
/// accessible via a TraceLevel static data member (Config.cs), to
/// send remote traces (if turned on), the client
/// must use the following code template:
///
///
///      Trace.WriteLineIf(Config.TraceLevel.TraceVerbose,
///              "starting packet translation...");
///
/// </summary>
///
/// <example>Trace.WriteLineIf(Config.TraceLevel.TraceVerbose, ///
starting packet translation...");</example>
/// <remarks>
/// To change the trace level you must edit the launching
/// applications configuration file as such:
/// The name attribute must match the name given to the trace
/// extension, e.g. RemoteTraceLevel
/// <system.diagnostics>
///            <switches>
///                  <add name="RemoteTraceLevel" value="1" />
///            </switches>
/// </system.diagnostics>
/// 0 (off), 1 (error), 2 (warning), 3 (info), OR 4 (verbose)
/// </remarks>
public class RemoteTraceListener : System.Diagnostics.TraceListener
```

```
{
   /// <summary>
   /// Reference to the remote tracing web service
   /// </summary>
   private RemoteTracerServer oRemoteService = null;

   /// <summary>
   ///
   /// </summary>
   public RemoteTracerService RemoteService
   {
         get { return oRemoteService; }
         set { oRemoteService = value; }
   }

   /// <summary>
   ///
   /// </summary>
   public RemoteTraceListener(string sName)
   {
         // initializes the remote web service that will receive the //
remote traces
         RemoteService = new RemoteTracerService();
         base.Name = sName;
   }

   /// <summary>
   /// Writes output remotely to the remote trace web service (trace
   /// receiver
   /// </summary>
   /// <param name="sMessage"></param>
   public override void Write(string sMessage)
   {
         RemoteService.RemoteTrace(Dns.GetHostName(), sMessage);
   }

   /// <summary>
   /// same as above
   /// </summary>
   /// <param name="sMessage"></param>
   public override void WriteLine(string sMessage)
   {
         RemoteService.RemoteTrace(Dns.GetHostName(), sMessage);
   }
```

You can see that the overridden Write and WriteLine methods simply call *RemoteService.RemoteTrace*, passing the original trace message. RemoteService, in this case, happens to be a Web service running on another system. This service

acts as the *trace receiver* and just so happens to be implemented as a Web service. Any receiver could have been created, as long as the RemoteTraceListener has access to it. I choose to implement the remote trace receiver as a Web service for simplicity. I could have opened a TCP/IP socket, for example, and could send the message directly to some socket server. How you implement the send or receiver is up to you.

Once your remote trace listener is constructed, you can now add it to your listener collection:

Listing 2.14: Sample routine for constructing and adding your custom listener.

```
public static void InitTraceListeners()
{
      . . .

      // for remote tracing
      if (Trace.Listeners[TRACE_REMOTE_LISTENER_KEY] == null)
            Trace.Listeners.Add(new
       RemoteTraceListener(TRACE_REMOTE_LISTENER_KEY);
}
```

Building a Remote Trace Receiver

Once added to the collection, any direct Trace.Write or Trace.WriteLine calls cause your corresponding methods to be called in RemoteTraceListener. Once called, the *RemoteTracerService.RemoteTrace* will be called. The RemoteTracerService is implemented as follows:

Listing 2.15: A sample Remote Trace Listener Web Service.

```
public class RemoteTracerService : System.Web.Services.WebService
{
      public RemoteTracerService()
      {
            InitializeComponent();
      }

      . . .

      /// <summary>
      /// Called by external clients to send all remote traces
      /// into a centrally supplied web service
      /// </summary>
      [WebMethod]
      public void RemoteTrace(string sSource, string sMessage)
```

```
      {
            EventLog oElog = new EventLog("Application",
                  Dns.GetHostName(), "RemoteTracer");

            try
            {
                  // first send it to the trace queue, queue
                  // should create itself if it has been deleted
                  Messenger oMessenger = new Messenger();
                  BusinessMessage oMessage =
                        oMessenger.MessageInfo;

                  oMessage.MessageText = sMessage;

                  // uri of the requesting party, this is the
                  // host name
                  oMessage.UserId = sSource;
                  // must set the message type we are looking
                  // for, this looks in the correct queue
                  oMessage.MessageType = "trace";

                  // send the trace to the queue
                  oMessenger.Send(oMessage);

                  // next send it a socket stream
                  string sRemoteTraceTargetHost = "etier3";
                  string sIP =
Dns.GetHostByName(sRemoteTraceTargetHost).AddressList[0].ToString();
                  int nPort = 8001; // any port will do
                  Utilities.SendSocketStream(sIP, nPort, sSource
                        + ":" + sMessage);

            }
            catch (Exception e)
            {
                  // or socket server was not listening, either
                  // way just log it and move on...
                oElog.WriteEntry(BaseException.Format(null, 0,
                        "Error Occurred During Remoting: " +
                        e.Message, e));
            }
      }
}
```

Sending Traces to a Message Queue

Inside the *try/catch* block, I demonstrate the sending of the trace message to two targets. The first message target is a message queue. The second target is any TCP/

IP listening socket server. I've wrapped the message queue interaction in a class called *Messenger* to help abstract the queuing services I may be using. For this example, the following source shows the main features of the Messenger class. This implementation of the Messenger uses the .NET *System.Messaging* library to communicate with *Microsoft Message Queuing*. Sending the trace messages to a queue is completely optional but it provides a quick means of persisting messages while providing an asynchronous delivery mechanism.

Listing 2.16: Sample Business Object to be placed on a queue.

```
/// <summary>
/// This is message information send/received from a queue
/// For example, for FTP file transfers the Data property will
/// contain the actual file contents
/// </summary>
public struct BusinessMessage
{
      /// <summary>
      /// see properties for each data member
      /// </summary>
      private string sType;
      private string sUserId;
      . . .
      private string sQueueName;
      private string sMessageText;
      private string sDate;
      private string sTime;

      /// <summary>
      ///   Specifying the type sets the queue name to
      ///   send/receive to/from
      /// </summary>
      public string MessageType
      {
            get {return sType;}
            set
            {
                  sType = value;
                  sQueueName = ".\\private$\\patterns.net_" +
                                    sType.ToLower();
            }
      }

      public string MessageText
      {
            get {return sMessageText;}
```

```
            set   {sMessageText = value;}
      }

      public string Date
      {
            get {return sDate;}
            set   {sDate = value;}
      }

      public string Time
      {
            get {return sTime;}
            set   {sTime = value;}
      }

      . . .

      public string UserId
      {
            get {return sUserId;}
            set {sUserId = value;}
      }

      public string QueueName
      {
            get {return sQueueName;}
      }
}

/// <summary>
/// Used for sending asynchronous messages to a durable message
/// queue of some kind
/// This currently using MSMQ and assumes it is installed,
/// eventually this will be implemented
/// to use any queuing framework.
/// </summary>
public class Messenger
{
      /// <summary>
      /// Data member for the main message information to be sent
      /// and received.
      /// </summary>
      private BusinessMessage oMessage;

      /// <summary>
      /// Property for the message information structure, which
      /// is an inner structure
      /// </summary>
      public BusinessMessage MessageInfo
```

```
{
      get {return oMessage;}
      set {oMessage = value;}
}

/// <summary>
/// Initializes an empty message information structure
/// </summary>
public Messenger()
{
      MessageInfo = new BusinessMessage();
}

/// <summary>
/// Sends the providing message info structure to a queue,
/// queuename should be set in the MessageInfo struct
/// This just set the MessageInfo property and delegates to
/// Send()
/// </summary>
/// <param name="oMessage"></param>
public void Send(BusinessMessage oMessage)
{
      MessageInfo = oMessage;
      Send();
}

/// <summary>
/// Sends the set MessageInfo data to the queue based on
/// the MessageInfo.QueueName property
/// This will be serialized as xml in the message body
/// </summary>
public void Send()
{
      try
      {
            string sQueuePath = MessageInfo.QueueName;
            if (!MessageQueue.Exists(sQueuePath))
            {
                  // queue doesn't exist so create
                  MessageQueue.Create(sQueuePath);
            }
            // Init the queue
            MessageQueue oQueue = new
                  MessageQueue(sQueuePath);
            // send the message
            oQueue.Send(MessageInfo,
                        MessageInfo.MessageType + " - " +
                        MessageInfo.Uri);
      }
```

```
        catch (Exception e)
        {
              throw new BaseException(this, 0, e.Message, e,
                                      false);
        }

  }

  /// <summary>
  /// Receives a message based on the MessageInfo.QueueName
  /// of the MessageInfo struct passed in.
  /// </summary>
  /// <param name="oMessage"></param>
  public BusinessMessage Receive(BusinessMessage oMessage,
                                 int nTimeOut)
  {
        MessageInfo = oMessage;
        return Receive(nTimeOut);
  }

  /// <summary>
  /// Uses the set MessageInfo.QueueName to retrieve a
  /// message from the specified queue.
  /// If the queue cannot be found or a matching
  /// BusinessMessage is not in the queue an exception will
  /// be thrown.
  /// This is a "polling" action of receiving a message from
  /// the queue.
  /// </summary>
  /// <returns>A BusinessMessage contains body of message
  /// deserialized from the message body xml</returns>
  public BusinessMessage Receive(int nTimeOut)
  {
        try
        {
              string sQueuePath = MessageInfo.QueueName;
              if (!MessageQueue.Exists(sQueuePath))
              {
                    // queue doesn't exist so throw exception
                    throw new Exception("Receive-Error"
                          + sQueuePath
                          + " queue does not
                          exist.");
              }

              // Init the queue
              MessageQueue oQueue = new
                    MessageQueue(sQueuePath);
```

```
                ((XmlMessageFormatter)oQueue.Formatter)
                      .TargetTypes =
                      new Type[]{typeof(BusinessMessage)};

                // receive the message, timeout in only 5
                // seconds -- TODO: this should probably change
                System.Messaging.Message oRawMessage =
                      oQueue.Receive(new TimeSpan(0, 0,
                      nTimeOut));

                // extract the body and cast it to our
                // BusinessMessage type so we can return it
                BusinessMessage oMessageBody =
                      (BusinessMessage)oRawMessage.Body;
                MessageInfo = oMessageBody;

                return oMessageBody;
          }
          catch (Exception e)
          {
                throw new BaseException(this, 0, e.Message, e,
                            false);
          }

    }

}
```

Sending Traces via Sockets

After creating the Messenger class, RemoteTracerService.RemoteTrace first retrieves the *MessageInfo* property to populate the message contents. The message contents are then populated and sent to the message using Send. From this point, we could return control back to the *trace originator*. However, for demo purposes, I also send the trace to a socket server on the network. I do this by setting my host, ip, and port, and calling my utility method *Utilities.SendSocketStream*. This method creates a connection with the specified host and if successful, sends the trace message as a byte stream.

Listing 2.17: Sample Socket Routine for sending any message.

```
public static string SendSocketStream(string sHost, int nPort,
string sMessage)
{
   TcpClient oTcpClient = null;
   string sAck = null;
```

```
   int nBytesRead = 0;
   NetworkStream oStream = null;

   try
   {
         oTcpClient = new TcpClient();
         Byte[] baRead = new Byte[100];

         oTcpClient.Connect(sHost, nPort);

         // Get the stream, convert to bytes
         oStream = oTcpClient.GetStream();

         // We could have optionally used a streamwriter and reader
         //oStreamWriter = new StreamWriter(oStream);
         //oStreamWriter.Write(sMessage);
         //oStreamWriter.Flush();

         // Get StreamReader to read strings instead of bytes
         //oStreamReader = new StreamReader(oStream);
         //sAck = oStreamReader.ReadToEnd();

         // send and receive the raw bytes without a stream writer
         // or reader
         Byte[] baSend = Encoding.ASCII.GetBytes(sMessage);
         // now send it
         oStream.Write(baSend, 0,  baSend.Length);

         // Read the stream and convert it to ASCII
         nBytesRead = oStream.Read(baRead, 0, baRead.Length);
         if (nBytesRead > 0)
         {
               sAck = Encoding.ASCII.GetString(baRead, 0,
nBytesRead);
         }

   }
   catch(Exception ex)
   {
         throw new BaseException(null, 0, ex.Message, ex, false);
   }
   finally
   {
         if (oStream != null)
               oStream.Close();
         if (oTcpClient != null)
               oTcpClient.Close();
   }
   return sAck;
}
```

Do not be overwhelmed by the amount of code shown here. If you haven't worked with either message queuing or the .NET network libraries, you can select another implementation. These transports are not a requirement. You could have simply written your trace message (once received by the Web service) to a file somewhere. This just shows a few options for being a little more creative and creating what could become a rather robust feature of your application's health monitoring. Keep in mind that whatever transport you use to display or store your trace messages can and will throw exceptions. Typically, exceptions should not be thrown back to the trace originator because that would defeat the purpose of tracing. *Remote trace errors should never halt a running application, and your code should take this into consideration*.

Building a Remote Trace Viewer

Now that we have a custom trace listener (to send trace messages remotely) and a trace receiver Web service (to receive trace messages remotely), how do we view them? Again, this is completely up to you. In the following code, I show a rather slimmed-down version of the production TraceViewer that is implemented in the *Product X* application we will cover in Chapter 6. In the following code, I use three System.Windows.Forms.DataGrid controls to display streamed, queued, and status messages. The controls are each bound to a grid using a System.Data.DataSet that is dynamically created in Visual Studio .NET. The DataSet was created from a corresponding XML schema file we created (see the technology backgrounder in Chapter 5 for information on XML schemas and automated DataSet creation). Two System.Threading.Timer objects are used to provide a threaded means of receiving our trace messages, both from a message queue and from a socket client. For a full listing, please download the complete source for the RemoteTraceViewer.

Listing 2.18: Sample Remote Trace Listener viewer (GUI).

```
public class frmSocketServer : System.Windows.Forms.Form
{
   . . .
   private static TcpListener oListener = null;
   private static Socket oSocket = null;


   . . .
   private System.Threading.Timer oQueuedTraceTimer = null;
   private System.Threading.Timer oActiveTraceTimer = null;
```

```
. . .
private System.Windows.Forms.DataGrid ActiveTraceGrid;
private static TraceInfo dsActiveTraceData = null;
private static TraceInfo dsActiveTraceDataCopy = null;
private static TraceInfo dsQueuedTraceData = null;
private System.Windows.Forms.DataGrid QueuedTraceGrid;
private System.Windows.Forms.DataGrid StatusGrid;
private static TraceInfo dsQueuedTraceDataCopy = null;

public frmSocketServer()
{

      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();

      frmSocketServer.dsActiveTraceData = new TraceInfo();
      frmSocketServer.dsActiveTraceDataCopy = new TraceInfo();

      frmSocketServer.dsQueuedTraceData = new TraceInfo();
      frmSocketServer.dsQueuedTraceDataCopy = new TraceInfo();

}

. . .

/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
static void Main()
{
      Application.Run(new frmSocketServer());
}

private void cmdListen_Click(object sender, System.EventArgs e)
{
      try
      {
            cmdListen.Enabled = false;
            cmdStop.Enabled = true;
            cmdRefresh.Enabled = true;

            if (chkEnableQueued.Checked)
                  oQueuedTraceTimer = new
                        System.Threading.Timer(new
                 TimerCallback(ProcessQueuedTrace), null, 0, 40000);
```

```
            if (chkEnabledStreamed.Checked)
                  oActiveTraceTimer = new
                        System.Threading.Timer(new
                 TimerCallback(ProcessActiveTrace),
                 null, 0,
                System.Threading.Timeout.Infinite);

            }
            catch (Exception err)
            {
                  EventLog.WriteEntry("…", "Trace Error: " +
                              err.StackTrace);
            }

      }

      private void cmdStop_Click(object sdr, System.EventArgs e)
      {
            StopListening();
      }

      /// <summary>
      /// Delegated Event Method for Timer to process Queued
      /// trace messages
      /// </summary>
      /// <param name="state"></param>
      static void ProcessQueuedTrace(Object state)
      {
            EventLog oElog = new EventLog("Application",
                  Dns.GetHostName(), "TraceViewMain");
            Messenger oMessenger = new Messenger();
            BusinessMessage oMessageIn = oMessenger.MessageInfo;
            BusinessMessage oMessageOut;

            try
            {
                  // must set the message type we are looking
                  // for, this looks in the correct queue
                  oMessageIn.MessageType = "trace";

                  while (true)
                  {
                    // grab the message from the queue
           WriteStatus(DateTime.Now.ToShortDateString(),
DateTime.Now.ToShortTimeString(), "Listening for trace messages on the
trace queue...");

                  oMessageOut = oMessenger.Receive(
```

```
                                oMessageIn, 10);

                    string sSource = oMessageOut.UserId;
                    string sDate =
                    DateTime.Now.ToShortDateString();
                    string sTime =
                    DateTime.Now.ToShortTimeString();
                    string sMessage = oMessageOut.MessageText;
                          sMessage = sMessage.Replace('\n', '-');


                    frmSocketServer.dsQueuedTraceData
                                .TraceMessage
                                .AddTraceMessageRow(
                                      sDate, sTime,
                                      sSource, sMessage);

                    frmSocketServer.dsQueuedTraceData
                                .TraceMessage
                                .AcceptChanges();

              }

        }
        catch (Exception e)
        {
              // exception was probably thrown when no message
              // could be found/timeout expired
              if (e.Message.StartsWith("Timeout"))
              {
        WriteStatus(DateTime.Now.ToShortDateString(),
DateTime.Now.ToShortTimeString(), "Timeout expired listening for
messages on trace queue.");
              }
              else
              {
                    oElog.WriteEntry("SocketServer - Error Occurred
                          During Message Receipt and Processing: "
                          + e.ToString());
              }
        }
  }

  private void StopListening()
  {
        cmdListen.Enabled = true;
        cmdStop.Enabled = false;
        cmdRefresh.Enabled = false;
```

```
        if (oSocket != null)
        {
        WriteStatus(DateTime.Now.ToShortDateString(),
                    DateTime.Now.ToShortTimeString(),
                          "Closing Socket ..." +
              oSocket.LocalEndPoint.ToString());
              oSocket.Close();
              oSocket = null;
        }
        if (oListener != null)
        {
        WriteStatus(DateTime.Now.ToShortDateString(),
        DateTime.Now.ToShortTimeString(),
                          "Stopping Listener ..." +
              oListener.LocalEndpoint.ToString());
              oListener.Stop();
              oListener = null;
        }

        if (oQueuedTraceTimer != null)
              oQueuedTraceTimer.Dispose();
        if (oActiveTraceTimer != null)
              oActiveTraceTimer.Dispose();
  }

  /// <summary>
  /// Delegated event method to process socket streamed trace
  /// messages
  /// </summary>
  /// <param name="state"></param>
  static void ProcessActiveTrace(Object state)
  {
        EventLog oElog = new EventLog("Application",
                    Dns.GetHostName(), "TraceViewMain");
        string sSource = null;
        int nDelimPos = 0;

        try
        {
              if (oListener == null)
              {
                    long lIP =
                          Dns.GetHostByName(
                    Dns.GetHostName()).AddressList[0]
.Address;
                    IPAddress ipAd = new IPAddress(lIP);

                    oListener = new TcpListener(ipAd, 8001);
```

```
                    oListener.Start();

        WriteStatus(DateTime.Now.ToShortDateString(),
                   DateTime.Now.ToShortTimeString(), "The server is
running at port 8001...");

WriteStatus(DateTime.Now.ToShortDateString(),
                   DateTime.Now.ToShortTimeString(), "The local End
                   point is  :" + oListener.LocalEndpoint);

                    while (true)
                    {  WriteStatus(DateTime.Now.ToShortDateString(),
                    DateTime.Now.ToShortTimeString(),
                    "Waiting for a connection on : "
                                  + oListener.LocalEndpoint);

                    oSocket = oListener.AcceptSocket();
        WriteStatus(DateTime.Now.ToShortDateString(),
                    DateTime.Now.ToShortTimeString(),
                    "Connection accepted from " +
                    oSocket.RemoteEndPoint);

                    // prepare and receive byte array
                    byte[] baBytes = new byte[1000];
                    int k = oSocket.Receive(baBytes);
                    WriteStatus(DateTime.Now.ToShortDateString(),
                    DateTime.Now.ToShortTimeString(), "Received
                    Message...");

                   // let's do it the easy way
                   string sReceivedBuffer =
                               Encoding.ASCII.GetString(
                               baBytes, 0, baBytes.Length);
                   WriteStatus(DateTime.Now.ToShortDateString(),
                   DateTime.Now.ToShortTimeString(),
                               sReceivedBuffer);

                   ASCIIEncoding asen = new ASCIIEncoding();
                   oSocket.Send(asen.GetBytes("trace received"));

                   nDelimPos = sReceivedBuffer.IndexOf(":");
                   sSource = sReceivedBuffer.Substring(0,
                      nDelimPos);
                   string sDate =
                         DateTime.Now.ToShortDateString();
                   string sTime =
                         DateTime.Now.ToShortTimeString();
```

```
                    string sMessage =
                          sReceivedBuffer.Substring(
                          nDelimPos + 1, sReceivedBuffer.Length -
(nDelimPos + 1));

                    sMessage = sMessage.Replace('\n', '-');


                    frmSocketServer.dsActiveTraceData
                                  .TraceMessage
                                 .AddTraceMessageRow(
                                     sDate, sTime,
                                     sSource, sMessage);

                    frmSocketServer.dsActiveTraceData
                                  .TraceMessage
                                  .AcceptChanges();
                    }
              }
        }
        catch (Exception e)
        {
              if (e.Message.StartsWith("A blocking operation"))
              {
         WriteStatus(DateTime.Now.ToShortDateString(),
              DateTime.Now.ToShortTimeString(), "Socket listener was
              canceled by the system.");
              }
              else
              {
                    System.Windows.Forms.MessageBox.Show(
                          "Error During Active Trace Listening: " +
                          e.Message);
                    oElog.WriteEntry(
                          "Error Occurred During Message Streaming"
                          + e.ToString());
              }
        }
        finally
        {
              if (oSocket != null)
              {

                    WriteStatus(…)
                    oSocket.Close();
                    oSocket = null;
              }
              if (oListener != null)
              {
```

```
                WriteStatus(…)
                oListener.Stop();
                oListener = null;
          }
     }
}

private void cmdClear_Click(object sender, System.EventArgs e)
{
     frmSocketServer.dsActiveTraceDataCopy.Clear();
     frmSocketServer.dsQueuedTraceDataCopy.Clear();
}

/// <summary>
/// Adds a new message to the status tab
/// </summary>
/// <param name="sDate"></param>
/// <param name="sTime"></param>
/// <param name="sMessage"></param>
private static void WriteStatus(string sDate, string sTime,
                              string sMessage)
{
     frmSocketServer.dsActiveTraceData
                    .StatusMessage
                    .AddStatusMessageRow(
                              sDate, sTime, sMessage);
     frmSocketServer.dsActiveTraceData
                    .StatusMessage
                    .AcceptChanges();
}

private void cmdRefresh_Click(object sender, System.EventArgs e)
{
      // copy active (streamed) grid's data and bind to grid
      ActiveTraceGrid.TableStyles.Clear();
      frmSocketServer.dsActiveTraceDataCopy = (TraceInfo)
          frmSocketServer.dsActiveTraceData.Copy();
      ActiveTraceGrid.DataSource =
                  frmSocketServer.dsActiveTraceDataCopy
                           .Tables["TraceMessage"];

      // copy queued (streamed) grid's data and bind to grid
      QueuedTraceGrid.TableStyles.Clear();
      frmSocketServer.dsQueuedTraceDataCopy = (TraceInfo)
        frmSocketServer.dsQueuedTraceData.Copy();
      QueuedTraceGrid.DataSource = frmSocketServer
                  .dsQueuedTraceDataCopy
                  .Tables["TraceMessage"];
```

```
        // same for status . . .

        // refresh all grids
        ActiveTraceGrid.Refresh();
        QueuedTraceGrid.Refresh();
        StatusGrid.Refresh();

        // this is absolutely required if you want to begin using
        // the grid styles in code
        // format the active trace grid
        if (frmSocketServer.dsActiveTraceDataCopy.Tables.Count > 0)
        {
              if (ActiveTraceGrid.TableStyles.Count == 0)
              {
                    ActiveTraceGrid.TableStyles.Add(new
                          DataGridTableStyle(true));

                    ActiveTraceGrid.TableStyles[0]
                                .MappingName = "TraceMessage";
                    FormatMessageGrid(ActiveTraceGrid, true);
              }

        }

        // format the queued trace grid
        if (frmSocketServer.dsQueuedTraceDataCopy.Tables.Count > 0)
        {
              if (QueuedTraceGrid.TableStyles.Count == 0)
              {
                    QueuedTraceGrid.TableStyles.Add(new
                          DataGridTableStyle(true));

                    QueuedTraceGrid.TableStyles[0]
                                .MappingName = "TraceMessage";
                    FormatMessageGrid(QueuedTraceGrid, true);
              }
        }

        // same for status

  }

  private void FormatMessageGrid(
                    System.Windows.Forms.DataGrid oGrid,
                    bool bIncludeSourceCol)
  {
        // if column styles haven't been set, create them..
        if (oGrid.TableStyles[0].GridColumnStyles.Count == 0)
        {
             oGrid.TableStyles[0].GridColumnStyles.Add(new
```

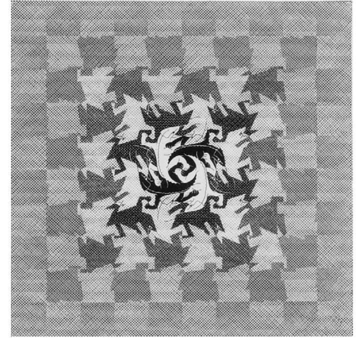
```
                  DataGridTextBoxColumn());
            oGrid.TableStyles[0].GridColumnStyles.Add(new
                  DataGridTextBoxColumn());
            oGrid.TableStyles[0].GridColumnStyles.Add(new
                  DataGridTextBoxColumn());
            if (bIncludeSourceCol)
      oGrid.TableStyles[0].GridColumnStyles.Add(new
                  DataGridTextBoxColumn());
      }

      oGrid.TableStyles[0].GridColumnStyles[0].Width = 90;
      // you must set each columnstyle's mappingname so that
      // other properties can be set since this is bound
      oGrid.TableStyles[0].GridColumnStyles[0]
                   .MappingName = "Date";
      oGrid.TableStyles[0].GridColumnStyles[0]
                   .HeaderText = "Date";

      oGrid.TableStyles[0].GridColumnStyles[1].Width = 90;
      // you must set each columnstyle's mappingname so that
      // other properties can be set since this is bound
      oGrid.TableStyles[0].GridColumnStyles[1]
                  .MappingName = "Time";
      oGrid.TableStyles[0].GridColumnStyles[1]
                  .HeaderText = "Time";

      . . .

      // format Source and Message columns styles …

}

private void cmdTest_Click(object sender, System.EventArgs e)
{
      try
      {
            throw new BaseException(this, 0,
                        "This is a test 1,2,3,4,5",
                        new Exception(
                              "this is the chained message",
                              null), true);

      }
      catch (Exception ex)
      {
            Console.Out.WriteLine(ex.Message);
            //do nothing
      }

}
```

To run the RemoteTraceViewer, select Listen. This will start the *timer threads*, which in turn call *ProcessActiveTrace* and *ProcessQueueTrace*. Each will listen and receive messages, and will add each message to a DataSet. To test your remote tracing, select the Test button. This will throw a nested exception, which will call your remote trace Web service. Once received, both timer threads should pick up the message and display it on each grid accordingly. Several hundred pages could probably be devoted to explaining the building of a production-ready remote tracing utility but this example should get you started in right direction.

SUMMARY

Exception handling is more than just throwing and catching objects in .NET. There are many design elements in providing a robust system, and providing a sound exception handling, logging, and tracing schema are some of the first steps. In this chapter, we covered several best practices for determining when to throw, catch, and ultimately log your errors.

I went into the differences of how exceptions should be handled in a Web service application. One of the most valuable components of this chapter is how you can exploit the FCL to provide global error log control through tracing. Finally, I provided a starting point for building a remote trace utility so that, as a service provider, you can receive complete details of a production system from anywhere on the Internet. The better your exception framework design, the less time you will spend in support mode, and the more time you will actually have to code. That should be incentive enough.



4

Middle-Tier Patterns

OVERVIEW

Earlier in the book we explained how middle-tier patterns don't necessarily dictate a physical deployment designation. In reality, any of the following patterns, whether they are considered design-oriented, architectural-oriented, or the like, can be implemented and physically deployed anywhere in an application. The middle-tier "category" does not necessarily predicate that these patterns belong on a server somewhere physically separated from a user client. When considering the location transparency of Web services, what makes up a middle tier can mean any logical combination of pattern, structure, or logic. The only really identifying characteristic among these patterns is the fact that they typically help implement components that:

- Do not provide any graphical or console interface characteristics (aside from debugging, e.g., using trace output to a debugger)
- House business rules
- Do not directly interact with persistent storage

That does not mean these cannot be applied to some graphical interface or even be part of a sophisticated data persistence design. These patterns simply belong to the category most likely to house the nonvisual business rules of the application.

Because middle-tier patterns cannot simply be considered strictly "design" patterns, they are more difficult to classify. Depending on how you implement them, they may have the ability to be part of several more traditional categories (see *Design Patterns—Elements of Reusable Object-Oriented Software*[1]—GoF). For example, the *Product Manager* pattern can be considered both *creational* and *behavioral*. The middle-tier classification only helps group a broadened combination of "middleware-friendly" patterns without having to give them a more traditional classification that you would find in these other sources of information. That is the point. We wouldn't have even organized them at all except that they become difficult to reference once there are several patterns to select from. For these reasons, the following patterns can be classified as middle-tier patterns.

Most of the following examples keep with the general solution "theme" and use our credit card processing application to drive the example code. We cover the following patterns in this chapter:

Chained Service Factory—Creating a single entry point for Web services

Unchained Service Factory—A late-bound single entry point for the Web services

Product Manager—Handling unmanaged code in a managed way

Service Façade—Delegating complex logic from Web services

Abstract Packet—Passing complex parameter sets into Web services

Packet Translator—Translating those complex parameters sets

---

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissades. Addison-Wesley, 1995. ISBN 0-201-63361-2.

With important (and related) mention to the following design patterns:

- Factory Method (GoF)
- Abstract Factory (GoF)
- Façade (GoF)
- Builder (GoF)
- Value Object (Sun)
- Proxy (GoF)
- Adapter (GoF)

CHAINED SERVICE FACTORY

Intent

Provide a single Web service method to allow the provider the flexibility of invoking different business functionality without directly affecting Web service clients. Provide a single entry point into a complex business framework.

Problem

Web service providers will eventually begin to provide Web services that go beyond accepting simple requests or providing simple responses. Not that Web services themselves need to complicated, but if they intend to be truly useful, the data sent to and from these services can become sophisticated. This pattern addresses the need for providing a Web service that not only performs a useful business function but also provides these services in a very flexible manner for invoking *any* business function. The single entry point is one of the first steps in defining what Microsoft has coined as a *service-oriented architecture*.

As providers begin to deploy these services, there will also be a need to change the logic behind those services. There needs to be a way to isolate the Web services client from these changes due to new features or altered implementations. Although no Web client can always be fully protected from future Web service functionality, the interfaces that they bind to should remain somewhat steady. The same goal of providing a standard interface "contract" to bind to (used in a more traditional implementation) can also be applied to the design of Web service

methods. The Web service method can be thought of as just another interface contract to adhere to and, thus, should be generic enough to facilitate those inevitable implementation changes. This is the distinct difference between services and APIs. Services, or more specifically, serviced-oriented architectures, should be based on messages, not remote procedure calls (RPCs). The trick to designing this type of architecture is to focus on being service-oriented and not object-oriented at this service entry point. You'll have plenty of opportunity to apply OO heuristics to the inner plumbing of your service architecture later.

We solve this problem the same way we solve the problem in a typical OO application—by fashioning a Web service method interface "contract." The interface contract will then maintain its definition by accepting the same set of parameters (including the return values) for each Web client. The Web client just needs the interface definition of the Web method to invoke any business function within the framework supported by this Web method. Each Web client would then be protected from future changes while still allowing the Web service provider flexibility in changing its function implementations.

The challenge in providing this generic interface-based Web service contract is that the parameters passed in the Web method also must either be generic or their type representations must be dynamic. They must be flexible and instructive enough to be able to describe the functionality that a Web client wants to call. In essence, the Web service method must become both a "factory" and a "delegator" for locating and calling the requested business functionality. The requested functionality must be described not by the Web method but by the contents of the generic parameters passed.

We implement this as a single Web service method with a single signature—*Execute()*. The parameters passed to Execute() provide the routing information or "factory inputs." By using a generic data type such as an ArrayList or a string containing XML, the Web service method can use those parameters to determine which business function needs to be created and called. The Web service method then becomes the "factory" for generating all requested business objects and becomes the single point of contact for all Web clients. In our example, all automated check processing or credit card authorizations function through the same Web service interface. The FinancialServiceFactory.Execute() is then called, whether it is a credit card customer requesting an authorization or a client requesting a credit report. Each request is funneled through the same interface and the same Web method. FinancialServiceFactory's job is to read the contents of the

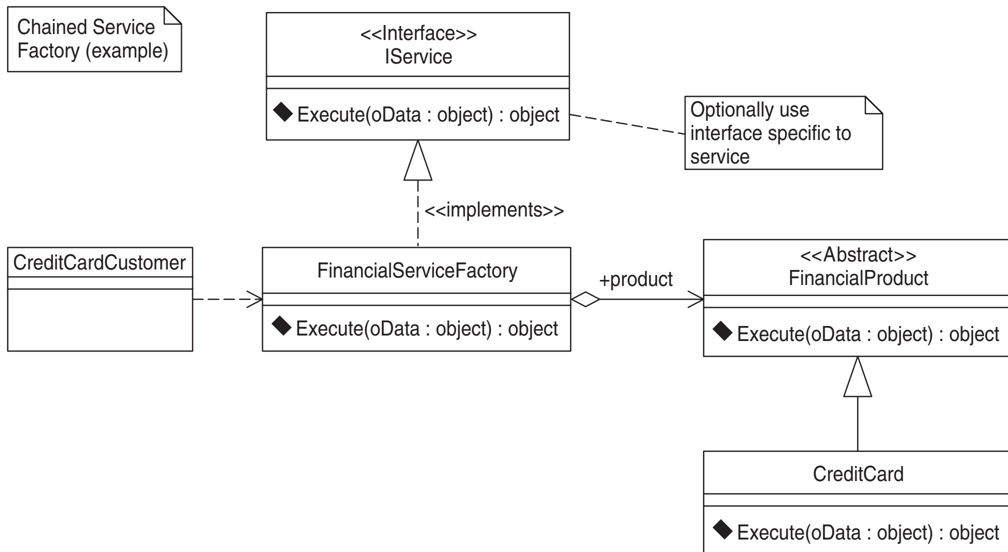


FIGURE 4.1: Service Factory implementation class diagram.

passed oData object, instantiate the appropriate FinancialProduct, and call the requested service. Figure 4.1 shows one possible implementation diagram of this pattern. To view the generic structure please reference Figure 4.2.

This presents another problem, however. What type is generic and dynamic enough to describe any functionality that each Web client may require of the Web service? If you are thinking XML, you are close. If you are thinking of XML schema definitions, you are even closer. The answer is the .NET DataSet. How did we arrive at that type? Well, DataSets are the perfect complement to a generic, type-safe data packager that not only understands an XML schema definition but also provides the developer with the facilities to manipulate them easily. See the Abstract Packet section later in this chapter; for an even deeper understanding, reference Chapter 5 for the technology backgrounder on XML schemas.

Forces

Use the Chained Service Factory pattern when:

- Business function interfaces may change in the future.
- Multiple Web service methods are provided on the server.

- Web service clients cannot be controlled.
- Changing client code would be difficult.

Structure

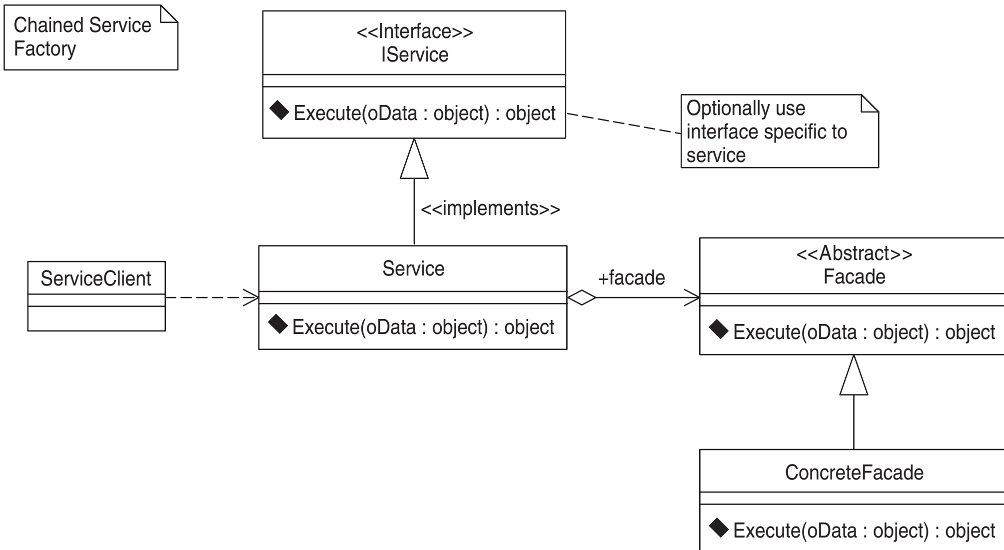


FIGURE 4.2: Service Factory generic class diagram.

Consequences

The Chained Service Factory has the following benefits and liabilities:

1. *It provides a single entry point into the system.* Due to the fact that only one method acts as the entrance into the framework, this allows greater control over who is calling into the system and more control over interface standards. It also provides a simple way of announcing services to the provider. Frameworks and services (especially within a large organization) are constructed all of the time. The key to selling them is providing a simple yet generic approach to calling the functionality.

2. *Eases system monitoring.* Because all traffic is routed through this single point of contact, monitoring the system becomes simpler. Using HTTP monitoring

applications (such as IIS Loader) simplifies the developer's profiling efforts because system load can primarily be determined through this one entry point.

3. *It isolates any Web client from Web method signature changes.* As stated in the above section, it gives a single entry point into the system and a generic signature that allows future services to be provided without affecting the external interface. This eliminates unnecessary Web service client proxy generations from occurring.

4. *It provides the ability to add future framework objects into the system.* Related to item 3, this also allows future subframeworks to be added easily into the system. For example, if all back-end business logic could be controlled by a primary controller object (see the Service Façade section in this chapter), that controller would be the instantiated target of the Chained Service Factory object. The Chained Service Factory would only need to be passed the service (from the client), at which time it would instantiate and delegate the business logic to that "controller." This places another level of abstraction and delegation into the model but it allows future "controllers" to be "plugged" into the system. This would allow not only changes to business functions but also additions to completely different sets of services. In our example, the primary set of business services involves credit card authorizations. For example, a scheduling system could then be built in the future. It could schedule any number of generic events, such as administration activities, batch reporting, etc. Adding another "free standing" scheduling façade into the system would now be much simpler. The Web client can then still activate this new set of functionality, using the same primary interface or entry point that was used for credit card authorizations.

5. *Calling* simple *Web service-based business functions requires more setup.* Although this design promotes the above benefits, it also introduces a bit of complexity when exposing "simple" functionality. Calling any Web services in this fashion involves preparing the parameters using a generic scheme. In our example, the data type I use happens to be a DataSet. This DataSet can be then populated with a least one row of information providing the metadata that can then be used to determine which business service and specifically which methods should be called. The metadata can be of any schema. That is the point. You now have the freedom to come up with any configuration of parameter data that you see fit.

Datasets do not have to be used but for our example, this seems to be the best and most flexible approach. The DataSet would be a single, generic, self-describing package that the Web client uses to request a specific back-end business function, using the Chained Service Factory as its entry point. The problem with designing such a sophisticated self-describing parameter is that now you are forcing the Web client to package its parameters for each and every call. This packaging occurs even for simple service requests unless you provide an entirely different Web method. For some cases, building a complex set of metadata commands and packaging them up just to call a simple business function via Web service may seem like overkill at times. It may be prudent to design a Web service method and separate interface for those cases. The designer must also take into consideration that passing complex types to Web service methods using a data type such as a .NET DataSet requires SOAP as the calling protocol. This means that the designer will not be able to test the Web service from a standard browser utilizing a simple HTTP GET or POST action and, thus, must create a custom test harness to test the Web service.

Participants

- ServiceClient (CreditCard Customer)—A Web service client for the credit card customer. This becomes the Web service proxy.

- Service (Financial Service Factory)—Contains a Web method that acts as a single point of entry into the system. This entry point unpackages the service request, instantiates the correct service (e.g., Service Façade), and calls a standard method on that service.

- Façade (Financial Product)—Defines a factory method to route the business request to the appropriate product. This is the entry point for each set of business functionality. It may contain all the business rules for a particular business category or may further delegate business behavior. This is usually an abstract or implementation parent that can contain logic such as data object construction or data preparation code (see the *Packet Translator* section in this chapter).

- ConcreteFaçade (Credit Card)—Implements the factory method for the business request. This entity optionally acts as a "controller" or Service Façade to other subordinate downstream business objects.

Implementation

The word *chained* in the pattern name comes from the fact that the instantiated service, which is created by the Web method, is early bound to that Web service. In other words, the Web method knows at design time what types it will be creating. When a credit card client calls Execute() on the FinancialServiceFactory, this method acts as a factory to any FinancialProduct derived class. The oData parameter passed can be of any generic data type, as long as it holds descriptive parameter data. Descriptive data refers to a form of "metadata" that can be used to describe what business functions the client wishes to invoke. The client's responsibility is to provide the metadata and package it is using with the rules defined by the Web method on the server. The metadata can take any form and be held using several generic types, as long as that type is flexible enough to contain this metadata and can be easily marshaled across the established invocation boundary.

In Listing 4.1, once the metadata is packaged and passed to the Web service, it is then used by the FinancialServiceFactory to instantiate the appropriate financial product—the CreditCard object in our example. The following code displays the signature of the Web method within the FinancialServiceFactory object. Notice the single parameter used to pass the data to the Web service. A .NET DataSet was chosen here due to its flexibility. A DataSet can then be used not only to contain this metadata but also to contain all of the actual data passed to the requested business function. The metadata can then be contained within a separate DataTable type as part of the DataSet. Other DataTables containing the actual data to be processed by the business function can also be passed without requiring the Web service to hold state between method invocations. All metadata and instance data is passed with one call to the Web method. The metadata is interrogated, the appropriate service is instantiated (see switch/case statement), and a standard method is called on that service. From that point on, it is up to the service (or FinancialProduct in our example) to disseminate it from there.

LISTING 4.1: Service Factory method sample implementation.

```
[WebMethod]
public DataSet Execute(DataSet dsPacket)
{
DataSet ds = new DataSet();
    FinancialProduct oProduct = null;
    string sService;
```

```
// call static translator method to extract the service
// name for this packet
    sService = PacketTranslator.GetService(dsPacket);
    switch (sService)
    {
          case Constants.PAYMENT_SVC:
                oProduct = (FinancialProduct) new CreditCard();
                break;
          case Constants.REPORT_SVC:
                oProduct = (FinancialProduct) new CreditReport();
                break;
          default:
                return ds;
    }
    // invoke the DoOp factory method on the facade
    ds = oProduct.Execute(dsPacket);
return ds;
}
```

Like any factory, a switch case statement is used to instantiate and early bind the CreditCard product to the FinancialServiceFactory. The financial product is determined in this example by a string value passed in the oData parameter. More specifically, the string is passed as part of a metadata DataTable as part of the passed-in DataSet (oData). The string is extracted from the Dataset's DataTable, as shown. A DAL (Data Access Layer) object is used to ease our work with the database (to see how the DAL is implemented, please reference Chapter 5. The data access specifics can be ignored for now; simply pay attention to the how the metadata column is accessed to extract the value of the service type. The returned *string* is then used by the *switch* statement above to determine which FinancialProduct to instantiate. Because both the CreditReport object and the CreditCard object inherit from the abstract type FinancialProduct, the lvalue in our case can be typed as the abstract type and its factory method called. Once Execute is then called on our instantiated FinancialProduct, that specific product can then disseminate the passed data as it sees fit. Listing 4.2 shows the helper method used to extract the service type from the incoming metadata.

LISTING 4.2: Service Factory metadata helper method.

```
public static string GetService(DataSet ds)
{
DAL oDAL = new DAL();
    string sValue = null;
```

```
// grab first and only row of the meta table passed as part of
// the dataset
    oDAL.Data = ds;
object oTemp = oDAL[Constants.META_TABLE, "SERVICE_COLUMN"];
    if (oTemp != null)
          sValue = (string)oTemp;
    else
          throw new Exception("…");

    return sValue.TrimEnd();
}
```

For example, if the Credit Card customer passes CreditCard as the service, it is used to instantiate the CreditCard class from the FinancialServiceFactory. Where this pattern differs from a typical factory method (GoF) is that it uses a Web service method to invoke the factory. It also differs in that it uses a generic data type to hold any of the parameters passed from any Web service client (a credit card customer, in this case). The DataSet is that generic type. A DataSet was chosen due to the fact that it is SOAP-friendly and very dynamic in that it can hold just about any structured data. Using DataSets provides the architecture with the most flexible alternative. Another option would be to use an ArrayList or even a generic custom object.

A Service Façade pattern can be used to add yet another level of abstraction between the Financial Service Factory and the FinancialProduct objects. Instead of using the metadata to instantiate a FinancialProduct, a Service Façade instead could then be instantiated and called. The mechanism is the same as it is in this example, just with an added level of abstraction. This would provide the ability to plug any service into the framework at any time without affecting bound Web service clients. For example, a schedule system façade could be implemented and added to the framework. This would then allow any Web client the ability to call this new "broad-based" service using the same calling semantics as calling for CreditCard services. The client simply has to package the metadata as before except that now it will be requesting a different service. The only code that would be required to change is the switch case statement setup in the Web service method. This change is required due to the fact that .NET is inherently early bound. The class type must be known ahead of time (the Service Façade object, in this case) and must be instantiated specifically based on the metadata passed in by the Web client. In the next section, I show you another pattern, the Unchained Service Factory, that eliminates this early bound requirement. This is possible

using .NET Reflection capabilities and will be explained in the upcoming technology backgrounder.

As you can see, this can be implemented with several levels of abstraction. The point and benefit to this is minimizing the impact on all of your current Web service clients while still providing your services the flexibility of adding new, more broadly based services to the framework at will.

Related Patterns

- Factory Method (GoF)
- Abstract Factory (GoF)
- Unchained Service Factory (Thilmany)
- Strategy (GoF)

UNCHAINED SERVICE FACTORY

Intent

Provide a single Web service method to allow the Web service provider the flexibility of easily interchanging the services without directly affecting bound Web service clients. Unlike the Chained Service Factory, however, this pattern uses .NET Reflection to implement late binding. Late binding allows the Web service the ability to instantiate any back-end business object without requiring code changes at the Service Factory level. Combining a factory object with the services of Reflection provides the design with a truly loosely coupled architecture on which to build.

Problem

This pattern solves the same problems as does the Chained Service Factory. The difference lies in the problem of forcing the developer to make changes to the switch/case code in the Service Factory class if any new business objects were to be added to the framework. Using Figure 4.3, if additional FinancialProduct derived objects were to be added to the system, the switch/case statement code in FinancialServiceFactory would have to change. The change would reflect the instantiation of the new class, based on that new service being requested by a complying Web client. As mentioned in the Chained Service Factory, business objects instan-

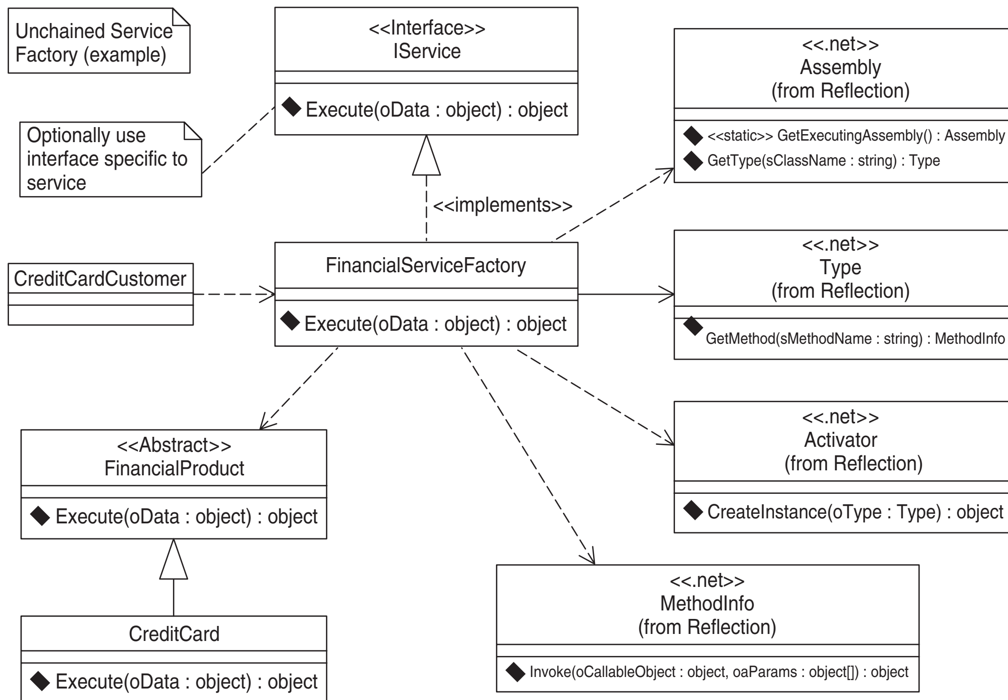


FIGURE 4.3: Unchained Service Factory implementation class diagram.

tiated by the "factory" in this manner are early bound to the service (thus the word *chained* in the pattern name).

Early binding means naming a type at compile time or design time (e.g., using the new keyword against a class)—in essence, building the vtable before execution. Late binding, on the other hand, refers to the technique of resolving a given type at runtime, as opposed to compile time. Late binding eliminates the need to know or name a type during design. It allows you to resolve and later reference a type and its members at runtime. This solution is then implemented in a much more dynamic and loosely coupled fashion. Although early binding will perform slightly faster in most respects, it does require recompilation of the code if any new features were to be added. Late binding protects the code from future deliberate recompilations. However, it does so at the penalty of slightly slower code performance. This is due to the fact that runtime type resolution generally will take longer than a vtable lookup. The tradeoff between slightly slower performing code versus increased flexibility must be weighed by you.

In the case of the Chained Service Factory, the added flexibility that is received by implementing late binding outweighs the slight decrease in performance. Using this pattern, however, only the initial instantiation and execution of the factory method is actually late bound. From the point of instantiation and initial execution onward, the remaining business object will be known and, thus, early bound. Also, the fact that there are only a few types needing to be resolved increases the benefit received from using late binding.

The Unchained Service Factory shows how using Reflection can aid you in eliminating code changes to the Service Factory. No longer does there need to be a fixed switch/case statement where each class must be predetermined and instantiated with the new keyword. The Web client requests the service as before by bundling the service request in some generic type, such as our DataSet. Instead of building of a switch/case statement, the developer can now extract the service name from the metadata and using that string, dynamically instantiate the target business object by name. Besides using .NET's Reflection classes (explained below), the designer also must name the targeted business objects using, at least in part, the same string that is requested by the Web client. Naming data types (classes, in our case) allows the developer to use text strings passed into the Web service method as our identifier of the class. This identifier is then used by the Reflection libraries to perform the runtime lookup and creation.

Anyone who has developed in Visual Basic, especially those who have built a similar factory using programmatic identifiers for this very purpose, should be already familiar with this technique. Using Visual Basic, a client could pass in a string of the programmatic identifier of the COM component they wished to create, and the receiving factory could then create that component on that value. Now that .NET no longer requires the use of programmatic identifiers (everything is based on name types), the type name itself must be used to provide this same feature. The difference here is that .NET is inherently early bound, and the developer must use Reflection to implement late binding.

Forces

Use the Unchained Service Factory pattern when:

- Business function interfaces may change in the future.
- Multiple Web service methods are provided on the server.
- Web service clients cannot be controlled.

- Changing client code would be difficult.
- Changes to a "chained" or early bound implementation of a factory would be inconvenient (to avoid having to edit code on a regular basis for all directly callable business functions).
- Late binding a publicly executable business function will not significantly impede performance.

Structure

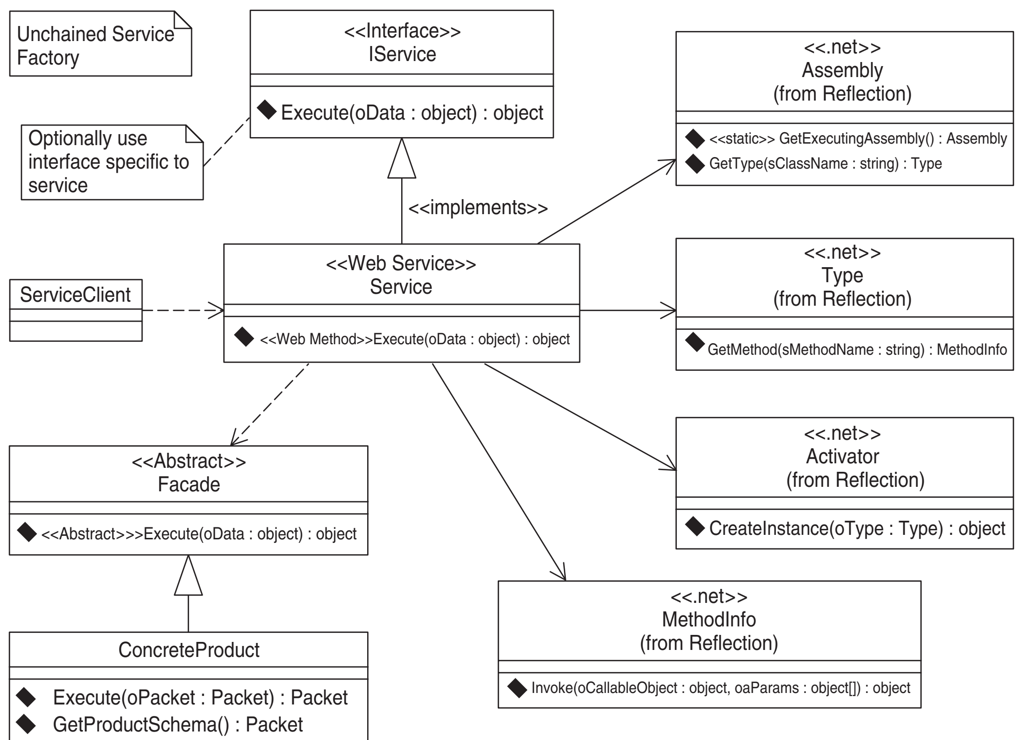


FIGURE 4.4: Unchained Service Factory generic class diagram.

Consequences

1. *Provides a true late-bound loosely coupled Web service.* Unlike the Chained Service Factory, the developer no longer must edit any code when business objects are added to the system. Using .NET Reflection services to invoke all initial

business objects, you can now easily add business objects into the system without forcing code compilations due to the factory having to early bind to any of the new business objects. This also isolates the Web clients from having to make any code changes on the client, even when calling significantly enhanced areas of the system, as long as the business objects added on the server conform to a standard contract and can be instantiated from the factory.

2. *Adds a small performance hit to the factory interface.* Using .NET Reflection services to instantiate the business object at that factory interface will slightly decrease performance. This is due to the fact that a business service must be "discovered" at runtime and, therefore, cannot be early bound. The impact should be minimal because all business services from the initially instantiated business object inward should be early bound. Only the initially instantiated business objects (FinancialProduct) or façades require reflection.

3. *Provides a single entry point into the system.* As stated in the above pattern, only one method acts as the entrance into the framework. This allows greater control over who is calling into the system and more control over interface standards. It also provides a simple way of announcing services to the provider. Frameworks and services (especially within a large organization) are constructed all of the time, but the key to selling them is providing a simple yet generic approach to calling them. With Reflection, any business function can be represented.

4. *Eases system monitoring.* As stated in the above pattern, because all traffic is routed through this single point of contact, monitoring the system becomes simpler. Using HTTP monitoring applications (such as IIS Loader) simplifies the developer's profiling efforts because system load can primarily be determined through this one entry point.

5. *It isolates any Web client from Web method signature changes.* As stated in the above section, it provides a single entry point into the system and a generic signature that allows future services to be added without affecting the external interface.

6. *It provides the ability to add future framework objects into the system.* Related to item 3, this also allows future subframeworks to be added easily into the system. See the Chained Service Factory section for details.

7. Calling *simple* Web service-based business functions requires more setup. See the Chained Service Factory section for details.

Participants

- ServiceClient (CreditCard Customer)—A Web service client for the credit card customer. This becomes the Web service proxy.
- Service (Financial Service Factory)—Contains a Web method that acts as a single point of entry into the system. This entry point unpackages the service request, instantiates the correct service (e.g., Service Façade), and calls a standard method on that service.
- Façade (Financial Product)—Defines a standard method to route the business request to the appropriate product. See the Chained Service Factory section for details.
- ConcreteFaçade (Credit Card)—Implements the standard method for the business request optionally acting as a “controller” or façade to other subordinate downstream business objects.
- Assembly—.NET framework class. This represents the currently running assembly from which to load the requested objects.
- Type—.NET framework class. This represents the type of the requested object.
- MethodInfo—.NET framework class. This represents the method of the requested service.
- Activator—.NET framework class. The object that creates the requested object once all reflection types are identified and created.

Implementation

Implementing this pattern is identical to that of the Chained Service Factory with one exception. Instead of building of what could eventually become a large switch/case statement (depending on how many business object types will be created), the developer must now use Reflection. Using the data types from the System.Reflection namespace, we now dynamically instantiate our FinancialProduct object or any other requested object by name. In the first step of the Execute() method, the service string is extracted as before in the Chained Service Factory. A reference to the current running assembly is then retrieved (explained below), and the business object type we are looking to create is returned. Retrieving this type requires the string we packaged in the DataSet (returned in the PacketTranslator.GetService()).

Once retrieved from the helper method, the string is concatenated to a fully qualified path that tells the reflection method GetType() which object we are looking for. Once the type is returned, we can create the object in memory by calling CreateInstance() using a Reflection "Activator." Don't worry, all of the Reflection material will be explained shortly. From that point on, we retrieve a method to call, bind our parameters, and invoke the method. The trick to calling any method in our newly created business object is guaranteeing that the method signature of that business object will always be consistent. Not providing a standard method signature on a creatable and callable business object in this design would significantly complicate the code. In fact, all business objects in this example implement a factory method themselves, which is called, you guessed it, *Execute()*. This is not a hard requirement, but it will make plugging in future business objects (that will be "launchable" from this Service Factory) much simpler and much cleaner.

Listing 4.3 shows this in action.

LISTING 4.3: Unchained Service Factory implementation using Reflection.

```
[WebMethod]
public DataSet ExecuteLateBound(DataSet dsPacket)
{
    DataSet ds = new DataSet();
    Assembly oPMAssembly = null;
Type oFacadeType = null;
    object oFacade = null;
    MethodInfo oFactoryMethod = null;
    object[] oaParams = null;
    string sService;

    // GetService code listed in ChainedServiceFactory section
sService = PacketTranslator.GetService(dsPacket);

// load the assembly containing the facades
    oPMAssembly = Assembly.GetExecutingAssembly();

// return the type and instantiate the facade class based
// on the service name passed
oFacadeType = oPMAssembly.GetType("CompanyA.Server" + "." +
    sService + "Facade");
oServiceFacade = Activator.CreateInstance(oFacadeType);

// Return the factory method for the chosen facade
// class/type
    oFactoryMethod = oFacadeType.GetMethod("Execute");
```

```
// bind the parameters for the factory method - single param
// in this case - dsPacket (DataSet)
    oaParams = new object[1];
    oaParams[0] = dsPacket;

    // invoke the Execute factory method of the chosen facade
    ds = (DataSet) oFactoryMethod.Invoke(oFacade, oaParams);
    return ds;
}
```

To understand the code completely, however, we need to dive a little into the .NET Reflection services.

Technology Backgrounder—.NET Reflection Services

For those who've been developing Java applications for some time, Reflection should be very familiar. In fact, .NET's implementation of its Reflection services matches that of Java's, almost feature for feature. For the rest of the folks who have not worked with any form of Reflection services before, this may seem a bit new. For COM developers, introspecting services through a set of standard interfaces is nothing new. Utilizing the basics of COM development and the QueryInterface() method, a developer can dynamically inspect the interface makeup of most COM components. Along those lines, using facilities such as type library information also provides a means for introspecting the types and services of binary entities. Reflection services provide exactly what the Java runtime intended—to provide a runtime programmatic facility to introspect and manipulate objects without knowing their exact type makeups at compile time.

Using types defined in the System.Reflection namespace in .NET, the developer can programmatically obtain metadata information about all data types within .NET. This can be used for building tools along the line of ILDasm.exe, where assemblies can be examined for their underlying makeup. Also, Reflection provides the ability to utilize late binding in an application that requires it (e.g., implementing our pattern). For those developing .NET/COM interoperability applications that implement late binding through IDispatch, this can also come in very handy. Using data types in the reflection namespace, one is able dynamically to load an assembly at runtime; instantiate any type; and call its methods, passing any of the required parameters. Instead of declaring those types at design time, the developer uses the abstracted data types included in Reflection. There are data types that represent objects, methods, interfaces, events, parameters, and so forth. To bind the generic types of Reflection with those that the developer actually

wants to work with, normal strings are used to name them at runtime. This may seem a little strange until you begin working with Reflection types, so the best suggestion is just to start writing a few examples. Some of the more useful members of the System.Reflection namespace are included in Table 4.1.

The first data type in the reflection namespace you need to familiarize yourself with is the Type data type. This isn't a misprint; the actual name of the data type is *Type*. Type is a class that provides methods that can be used to discover the details

TABLE 4.1: Some Members of the System.Reflection Namespace

Assembly	Define and load assemblies, load modules that are listed in the assembly manifest, and locate a type from this assembly and create an instance of it at runtime.
Module	Discover information such as the assembly that contains the module and the classes in the module. You can also get all global methods or other specific, nonglobal methods defined on the module.
ParameterInfo	Information discovery of things such as a parameter's name, data types, whether a parameter is an input or output parameter, and the position of the parameter in a method signature.
PropertyInfo	Discover information such as the name, data type, declaring type, reflected type, and read-only or writeable status of a property, as well as getting/setting property values.
EventInfo	Discover information such as the name, custom attributes, data type, and reflected type of an event. Also allows you to add/remove event handlers.
FieldInfo	Discover information such as the name, access modifiers (e.g., public), and the implementation details of a field, as well as getting/setting field values.
MethodInfo	Discover information such as the name, return type, parameters, access modifiers, and implementation details (e.g., abstract) of a method. Use the GetMethods or GetMethod method of a Type object to invoke a specific method, as we do in this example.

behind other data types. However you cannot directly call "new" on the Type class. It must be obtained through another "type-strong" data type, such as our Product object below:

```
Product oProduct = new Product();
Type t = oProduct.GetType();
```

Another technique is to do something like this:

```
Type t = null;
t = Type.GetType("Product");
```

Something you will see quite often when using attributes or any other method that requires a System.Type data type—getting a Type object using the typeof() keyword:

```
Type t = typeof(Product);
```

Once we have a Type object, we can get to the objects methods, fields, events, etc. For example, to get the methods (once we have the Type object), we call t.GetMethods() or t.GetFields(). The return values of these calls return other data types from the Reflection namespace, such as MethodInfo and FieldInfo, respectively. Hopefully, you're starting to get the picture.

Finally, to get our pattern working, we first retrieve the current assembly by calling GetExecutingAssembly(). This retrieves an Assembly object so that we can later return a Type object representing a data type requested by the Service Factory:

```
    Assembly oPMAssembly = null;

/ load the assembly containing the facades
oPMAssembly = Assembly.GetExecutingAssembly();
oFacadeType = oPMAssembly.GetType("NamespaceA.ProductFacade");
```

Once we have the assembly, we can retrieve the business object using a normal string, such as the one passed to us in the Service Factory method Execute(). Once we have a returned Type object, we can now actually instantiate the object. This requires using what is called the *Activator class* in Reflection. The Activator is the key to implementing late binding in .NET. This class contains only a few methods, one of which is called CreateInstance(). Here we pass our newly returned Type object, which results in the runtime creation of our desired busi-

ness object. Once our object is created, the next step is to bind parameters and retrieve a MethodInfo object to call. This is accomplished by first binding the method parameters as an object array, retrieving a MethodInfo type by calling GetMethod("<method to call>"), and finally calling Invoke on that newly returned MethodInfo type, as shown:

```
oFactoryMethod = oFacadeType.GetMethod("Execute");

oaParams = new object[1];
oaParams[0] = dsPacket;

// invoke the Execute factory method of the chosen facade
ds = (DataSet) oFactoryMethod.Invoke(oFacade, oaParams);
```

That's it. Now the developer can pass any data type string into the Service Factory, and as long the requested business object implements a standard interface, any object can be created. This is just the tip of the iceberg for the Reflection services, and I strongly urge you to explore other features of this powerful namespace for yourself. Be forewarned, however, that because there will be some performance penalty when using Reflection, it should be used sparingly.

Related Patterns

- Factory Method (GoF)
- Chained Service Factory (Thilmany)
- Proxy (GoF)
- Abstract Factory (GoF)
- Strategy (GoF)

Challenge 4.1

When would you implement both Chained and Unchained Service Factory patterns as part of the same architecture?

See Chapter 6 on Product X to see why they did it (pages 303–309).

PRODUCT MANAGER

Intent

Provide a framework to which to migrate unmanaged code. Help isolate clients from semantically different technologies. Control "unmanaged" code in a more managed fashion.

Problem

When building a business services framework in .NET, unless you are lucky, you will have to support some form of legacy services (e.g., DLLs, COM components, or any unmanaged code). It is strange having to refer to existing COM components as legacy. This is especially true because COM+ is still used with .NET. However, when dealing with unmanaged pieces of code, more care should be given. Whether it is an existing framework that you are migrating from or some other third-party application that you must integrate, this issue will be there. The trick is not figuring out the bridging technology upon which to call these legacy services but how to still provide a clean design. How does one design a set of managed .NET code when some of those services will be using unmanaged services? How does one isolate the client from knowing or caring that the services it requests are actually being handled by unmanaged code?

For starters, abstracting a calling client from the back-end business services will help. This will isolate the technical differences apparent only in the object bridging those services. Those who have used the Abstract Factory (GoF) or Strategy Pattern (GoF) should be very comfortable with this design approach. The Product Manager takes a step further by combining these patterns (so to speak) to form a design that will not only provide a standard contract upon which to call any back-end business service but also a level at which to isolate any code that will act as the "bridge" between managed and unmanaged code. The main controller of any services exposed by either managed or unmanaged code is the client of the Product Manager. This client takes the form of a façade object in our example. Here, in the PaymentFacade, we control which product to instantiate and, using contracted methods on each product, ask the Product Manager to execute the requested service (Figure 4.5). The PaymentFacade object acts as the controller to the ProductManager. It interacts only with the "contracted" interface provided by this abstract base class. All of the common business implementation code for this design is

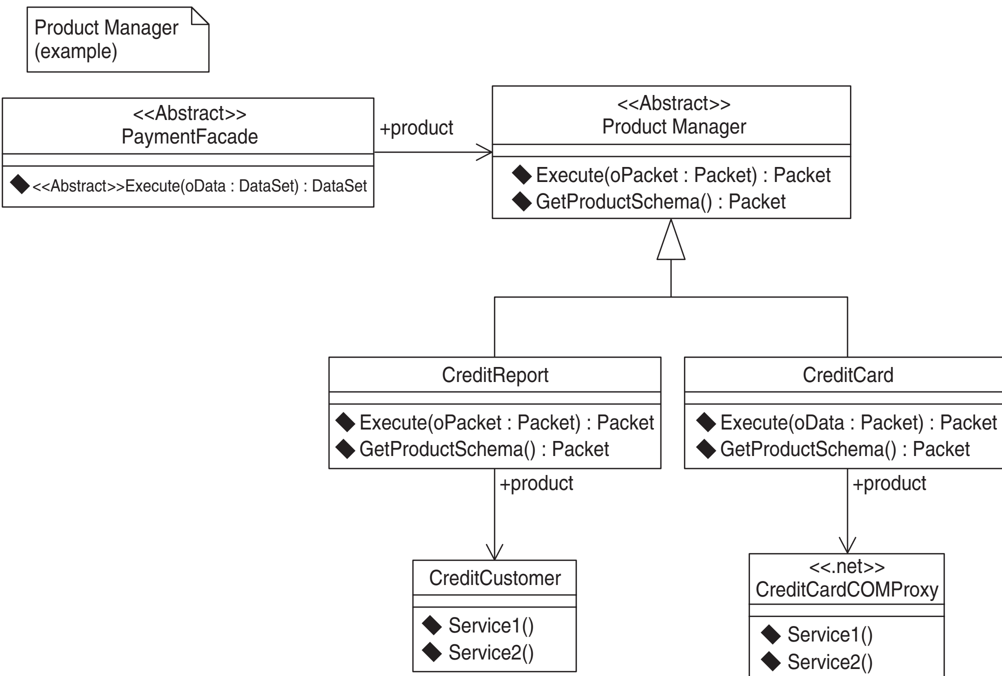


FIGURE 4.5: Product Manager implementation class diagram.

located in the ProductManager base class. Any of the product-specific implementation code resides in the child classes of the Product Manager. Specifically, code to handle unmanaged versus managed semantics will be located in these child classes.

For example, let's say you wish to use XML schemas to describe each underlying packet that is passed from object to object. Each packet can also be passed into each product class. To provide such a feature, the designer may wish to store that schema in a database and load a packet from that schema. This is something you will see in Chapters 5 and 6. Retrieving that schema from the database will be done in the same exact way, no matter which product child class the controller is talking to. Hence, this is some of the common code we are referencing. All the controller needs to know is that a schema must be retrieved to hydrate a packet. The common code for retrieving this schema can be easily placed in an abstract parent ProductManager class. This is part of the code common to both unmanaged and managed code.

Each product child object merely has to override methods that provide the base ProductManager class with the key to use to retrieve these schemas. All the database logic code still resides in the base class of the abstract ProductManager parent. Code specific to each product resides in the child product classes. This is object orientation 101. The difference is that all product-specific code to call unmanaged or unmanaged code remains in the child classes. The unmanaged product child class still benefits from any of this common code while still isolating the controller from any the differences through the use of the abstract parent.

For a product that must bridge older technologies such as a COM component, a product child class simply acts as the calling proxy to the underlying COM code. Using the model below, a CreditCard product acts as a normal client to the COM proxy code generated with the .NET/COM interoperability tools. Any COM-specific package or unpackaging occurs in this CreditCard product object. The CreditReport object is a standard common language runtime (CLR)-managed object. It can call other .NET help objects or be self-contained; it doesn't matter. The point is, neither the client nor even the controller class know or care that the back-end business functions running may be unmanaged code. This design also provides the unmanaged code access to common managed services. All in all, this pattern allows common .NET-friendly code to be shared in an abstract base class while isolating product-specific code in the derived product child classes.

Forces

Use the Product Manager Pattern when:

- Migrating old business services into a new technology platform.
- Bridging COM services from a .NET framework.
- Calling semantically different business services from a single interface, such as when using some form of "factory" pattern (e.g., Abstract Factory, Chained Service Factory, etc.).
- You want to leverage common code yet isolate technology-specific code from a common caller.

Structure

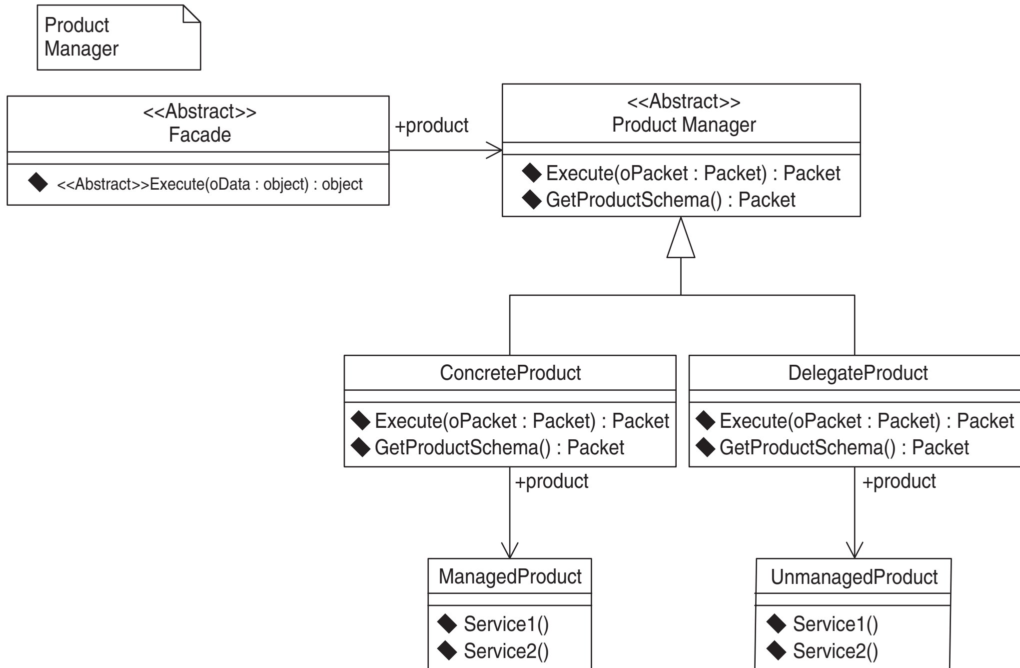


FIGURE 4.6: Product Manager generic class diagram.

Consequences

1. *Isolate technology-specific implementation details.* When building a framework that will employ calling unmanaged code, most likely there will be technology-specific nuances that will be included, for example, if the unmanaged code provides business services that would initially take too long to migrate fully over to the unmanaged world. The best solution would be to leverage off those services from the managed code. Typically, this is done with a bridge or proxy. Doing so may incorporate technology-specific details. However, that should not affect the "managed" architecture. If these unmanaged services were to be COM+ components, there may be some technology-specific attributes. These could include those that control transaction semantics or other COM+-specific attributes. Abstracting these details will help isolate calling clients and help decouple technology semantics within a framework aimed at creating managed-friendly semantics. The abstraction can be easily implemented using an interface imple-

mentation contract defined by the managed framework. Taking it a step further, an abstract class (ProductManager) can then be created to house common code that both managed and unmanaged products can share. All technology-specific code can then reside in the derived product classes, isolating the Controller Class (Façade, in our example) from implementation details.

2. *Provide a migration path to the new .NET framework.* Fully reengineering legacy business services is a luxury most development teams do not have. Leveraging those business services as much as possible usually is the best answer. Providing a migration path to the “new managed world” then seems to be the best architectural solution. By isolating technology details as mentioned in the previous consequence, the Product Manager provides a migration path to this managed world. External legacy services can then be incorporated into the architecture by deriving themselves by an abstract parent class that may contain managed code common to all services. The derived product class can then simply act as a client to the external services, as any other client would act if it were not implemented using managed code. The designer can then slowly migrate functionality to this derived product class without affecting common code.

3. *Leverages common managed code for managed calling proxies to unmanaged code (that’s a mouthful).* Building a proxy to unmanaged code should also provide access to the common architecture code to which Product Manager objects have access. By implementing an abstract base class and placing all common managed code here (the ProductManager), the managed proxy to the unmanaged code will, through inheritance, still have access to it. This will promote leveraging as much of the new architecture as possible, even when calling legacy services such as COM components.

Participants

- Façade (PaymentFacade)—“Controller” class that acts as a client to the Product Manager. The façade, in this case, speaks only “Product Manager language.” This simply means only public interfaces exposed by the Product Manager and implemented by the specific products will be callable from the façade.

- ProductManager (Same name as implementation)—The abstract base class for each product object in the design. There can be a Product Manager for each category of product. This contains the standard interface “contract” (abstract

functions) that must be implemented by each derived product class. It also contains all code common to all products in the class tree.

- ConcreteProduct (CreditReport)—Contains all code for the business logic on the managed code side. This can be self-contained or it may aggregate other downstream business objects. It must implement all abstract members of the ProductManager to conform to the Product Manager's contract.
- ConcreteDelegate (CreditCard)—Contains all code for the business logic on the unmanaged code side. This usually acts as the gateway to the external service or unmanaged code. It must also implement all abstract members of the ProductManager to conform to the Product Manager's contract.
- ManagedProduct (CreditCustomer)—Optional helper class used to implement the managed business code fully.
- UnmanagedProduct (CreditCardCOMProxy)—This represents either the proxy code or the third-party services. In the case of calling COM components, this is the generated proxy code (using tblimp.exe utility or VS.NET). Whether a generated proxy or third-party library, this is any unmanaged code that the architecture wishes to leverage.

Implementation

One of the benefits of implementing the Product Manager pattern is its general applicability. Like similar patterns, such as Abstract Factory and Factory Method, this pattern can be applied to many scenarios through the architecture. There can be several difference Product Manager abstract classes. If the designer so chooses, there can be a single common Product Manager class from which to derive all product classes. The ProductManager class can be driven directly from a GUI-based client or from another business object, such as a controller or façade. In our example, we use a Façade object to manage the ProductManager abstract class. The PaymentFacade participates in the creation of the appropriate Product class (CreditCard and CreditReport, both derived from ProductManager). Because each Product class derives and implements the "contract" for the Product Manager, the façade calls only those interfaces exposed by the ProductManager abstract class (e.g., Execute()), as shown in Listing 4.4.

LISTING 4.4: Product Manager Factory Method implementation.

```
public Packet Execute(Packet oPacket)
{
      ProductManager oProdMan;

    switch (oPacket.Type)
      {
          case CREDIT_CARD_TYPE:
              oProdMan = (ProductManager) new CreditCard(oPacket);
              break;
          case PMConstants.CHECK_TYPE:
              oProdMan = (ProductManager) new CreditReport(oPacket);
              break;
          default:
              oProdMan = null;
              break;
      }
      return oProdMan.Execute();
    }
```

In essence, this code is acting as another factory on each product. In addition, it is calling the established "strategy" style interface to invoke the action on the product (see the Strategy Pattern section from the GoF). Once invoked, the product is able to disseminate the action by either further delegating the call or by handling the invoked business function immediately. In the "eyes" of the Execute() statement above, only the signature of the Product Manager is known. The fact that CreditCard is delegating to the proxy COM wrapper is unknown. Both CreditCard (forwards to unmanaged code) and CreditReport (forwards to managed code) look the same to the façade in this scenario. When Execute() is finally called on the CreditReport object, it looks something like as shown in Listing 4.5.

LISTING 4.5: Product Manager delegation implementation.

```
public override Packet Execute()
    {
          Packet oPacketOut;

          switch (Packet.Action)
          {
                case AUTHORIZE_ACTION:
                      oPacketOut = Authorize();
                      break;
                case CAPTURE_ACTION:
                      oPacketOut = Capture();
```

```
                    break;
                case GET_PRODUCT_SCHEMA_ACTION:
                    oPacketOut = GetProductSchema();
                    break;
                default:
                    oPacketOut = null;
                    break;
            }
            return oPacketOut;
}
```

Here you see that, like the Façade object, the Execute method of the CreditCard product simply determines (by using another property of the packet) which business function this should delegate to. If the packet requests an "authorization," this will call a local method, Authorize(). Authorize() will delegate to a proxy object that was generated via tblimp.exe. The authorization will actually occur in unmanaged code, and the CreditCard product object is simply acting as a client to that code. This not only provides a migration path from COM services already written, but it also isolates the controller or Façade object from knowing when this is handled in unmanaged or managed code. The Authorize() method looks something like this (Listing 4.6):

LISTING 4.6: Sample Product Manager worker implementation.

```
public Packet Authorize()
    {
            Packet oPacketOut = new Packet();
            // COM Proxy object

            COMAuthorizer oAuthorizer = new COMAuthorizer();
            // .. build COMPacket from Packet
            oAuthorizer.DoOp(COMPacket, null);
            // .. build Packet from returned COMPacket
            return oPacketOut;
    }
```

Implementing the other product object, CreditReport in our example, is exactly the same. The difference is that CreditReport contains all managed code. CreditReport and CreditCard can each leverage common code contained in the abstract ProductManager. However, both can still contain technology-specific implementation details hidden from the outside world.

Related Patterns

- Factory Method (GoF)
- Chained Service Factory (Thilmany)
- Unchained Service Factory (Thilmany)
- Virtual Proxy (Grand)
- Abstract Factory (GoF)
- Strategy (GoF)

CHALLENGE 4.2

What other kind of products could utilize the Product Manager besides what is shown here?

See Chapter 6 on Product X to see (pages 296–303).

SERVICE FAÇADE

Intent

Encapsulate business logic using a "controller" class to become the primary entry point from which all business logic (usually of a specific business category) will be contained and/or driven. Provide Web services with single entities to call directly.

Problem

For this pattern, probably the simplest of all mentioned patterns in this book, the problem solved is also simple. When developing Web services in .NET, developers may immediately get the tendency to begin placing business logic within the code directly behind the Web service class provided by VS.NET. This includes any Web methods added to a Web service. The problem here is that, as you may already surmise, "cluttering up" this section with business rules and complex logic can soon become very difficult to manage. Another problem with this sort of "develop before design" approach is that if you desire to reuse those services elsewhere, it will always force the calling client to go through the Web service to do it. The architecture actually may "require" that all back-end business functions be called from a single entry point, such as a Web service. This is probably due to security and control. However, this may not be the most efficient means of invoking busi-

ness services. Those who have already been developing Web services realize that there is a performance penalty when invoking a business function through a Web service if called as a standard Web service client.

To provide the "best of both worlds," that is, to provide an optional means to force all traffic through a Web service while also allowing other means of invoking controlled business functions, a Service Façade should be used. The Service Façade, like a traditional façade, simply provides a "controller" object to expose all public "callable" business functions to any external clients. The Service Façade becomes the server to the actual code directly behind that of the Web service and its Web methods (Figure 4.7). Instead of cluttering the implementation directly behind the Web method, this simply delegates calls to the appropriate Service Façade object. This also provides another means of invoking the business functions without forcing the client to go through the Web service or act as a Web service client. The Service Façade could be called from a standard ASP.NET implementation or another server component.

The difference between a traditional façade pattern and a Service Façade is "architectural," meaning that the Service Façade must take into account the possibility of both Web methods acting as a client to the façade and other non–Web-service clients. This is implementation-specific. For example, exceptions thrown from the Service Façade should take multiple clients into account, as well as provide a Web service-friendly error-handling mechanism. In some cases, you should avoid throw-

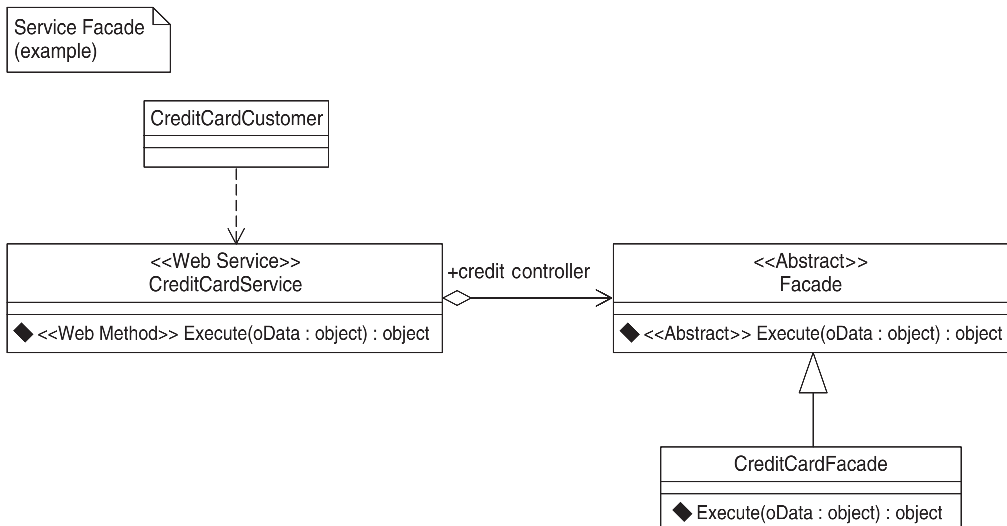


FIGURE 4.7: Service Façade implementation class diagram.

ing exceptions altogether or in some designs do so when transactions are logged. The point is that Web service clients may need to be taken into account at this level.

Forces

Use the Service Façade Pattern when:

- Implementing complex business logic that may be called externally.
- Servicing multiple clients from a Web service.
- Code becomes too complex to place directly into a Web service class.
- The design calls for categorizing business functionality that may contain different means of handing architectural features, such as exception handling, logging, security, etc.
- Web service class inherits from System.Web.Services.WebService and must also inherit from another base class in order to receive additional functionality, such as when needing to derive from System.EnterpriseServices.ServicedComponent. (This is the case when the controllers become COM+ components, because multiple-implementation inheritance is not supported in .NET.)

Structure

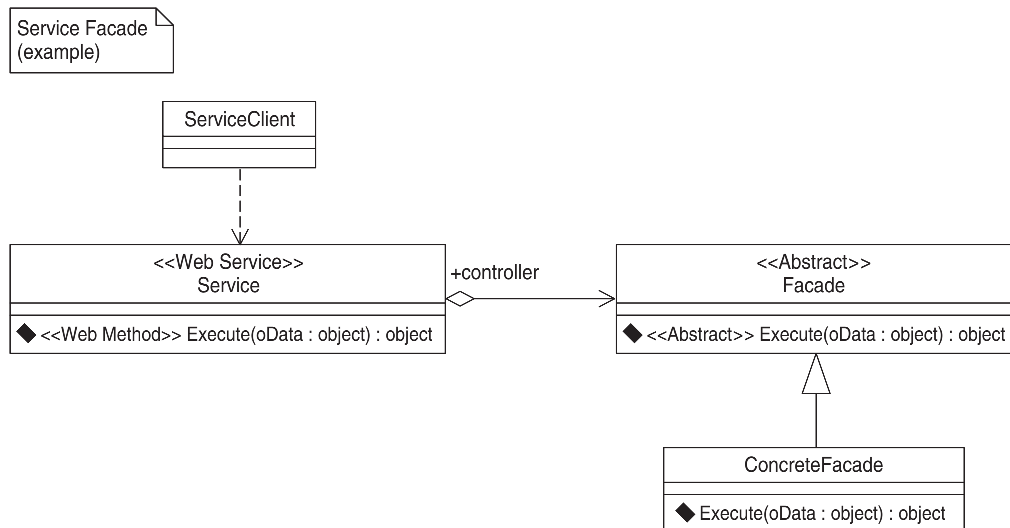


FIGURE 4.8: Service Façade generic class diagram.

Consequences

1. *Eliminates having to place complex logic within the language implementation of a Web service or ASP.NET file directly* (aspx or asmx source). Instead of "cluttering up" the source code directly within a Web method, the developer should utilize a Service Façade class. A Service Façade is strictly a façade that coordinates and houses complex business logic for a specific business category.

2. *Provides manager for specific business areas.* A Service Façade also acts like a traditional Façade pattern in that it manages specific business areas, providing high-level business functions typically exposed to the public. The exposed business functions in this case will be accessible to the Web service. A Web method can be defined for each publicly exposed façade function, or functions can be grouped into single Web method. Using a single Web method was explained in the Service Factory sections in this chapter.

3. *Provides an entity in which to house Web service-specific exception handling*. When throwing exceptions to external calling clients, the architecture must take into the account the tier that will be directly callable. In the case of the Service Façade, any Web service clients will be indirectly calling the Service Façade because the Web service stands between the client and the Façade class. In order for some clients to receive rich error-handling adjustments, there may need to be adjustments made to the architecture, depending on where the client is located.

4. *Provides an entity in which to house Web service-specific parameter passing*. Along the same lines as the above point, passing data between tiers physically located on the same machine could be quite different. There may be architectural scenarios, such as when the calling protocol uses SOAP, where changes to the parameter passing scheme need to be made. These architectural adjustments could be housed within the logic of the Service Façade. Neither the callable business object within the system nor the Web service class itself should be cluttered with this type of logic. Doing so will provide a more reusable design.

Participants

- Service (CreditCardService)—This is any Web service containing Web methods. Each Web method calls a requested exposed business method on the façade. This can be specific methods or simply (as in our case) a single interface

method (e.g., Execute()). The Web service simply becomes the direct interface to the Web client. The Web method does not contain any business-specific functionality. The business rules are all delegated to the ConcreteFacade class or any of its optional parent entities.

- Façade (Same name as implementation)—Base class for any façade classes that act as a controller to any service. The façade, in this case, speaks only the "language" of the business as it publicly exposes high-level business functionality. This simply means that only public interfaces are exposed to its Web method clients. Web service client requests are delegated from a Web method to the Service Façade.
- ConcreteFacade (CreditCardFacade)—The main implementer of the publicly available business interface. This class could be self-contained or could delegate to other specialized business objects. Most business rules are contained or driven from here, especially those related to packaging data that must be sent back to the Web service client.

Implementation

The beauty of the Service Façade pattern is that is it simple. It requires more forethought only when it begins to provide the facilities to take into account its Web service clients (as mentioned earlier). Such facilities include handling exceptions or parameters in special ways. However, features such as these are implementation-specific. How to design exception handling or data passing is really another topic and is covered in Chapter 2. The Service Façade, however, provides the construct within which to build these features so that once again, the code directly "behind" the Web methods remains clean, and the business services housed by the Service Façade remain reusable. Implementing the Service Façade is very similar to any of the controller or manager classes that I'm sure you have designed in the past.

Keep in mind that the Service Façade, like a traditional façade (GoF), is the driver of publicly available business logic. It should require more forethought if you are designing a more complicated system (no kidding, right?). The point is, if there will be several categories of business functionality, different approaches should be considered. One implementation approach would be to use an abstract class or interface from which to derive each façade. In fact, this is what our example uses. This is certainly not a requirement and shows only one implementation. Another approach is to create a full implementation-inheritable base class for all façades.

The choice is yours and depends more on how the Service Façade itself will be implemented. For example, if you building a mission-critical system that must support multiple clients, shared resources, and possible transaction management, then implementing your Service Façade classes as COM+ components may be a wise choice. If this is the case, you should use either a simple interface or a full base class as the parent to your Service Façade. The reason is that .NET does not support multiple inheritance. Deriving from more than one base class is not permitted unless the class you are also deriving from is an interface. If you are a Java programmer, you are already familiar with this rule. Because adding COM+ features requires inheriting from the System.EnterpriseServices.ServicedComponent, you could make the parent Façade class inherit from it, thus gaining this functionality for each Service Façade child class. You could if your parent Façade class was an abstract class and you still wanted to derive from another base class, but it would seem odd and would not (considered by some) be the cleanest of designs. For our example, I use an abstract base class with the option of knowing that this could be relatively easy to change to an implementation base class if I so desired. We could then incorporate COM+ features by using ServicedComponent as its base class. For now, let's just stick with an abstract parent class to the Service Façade (Listing 4.7).

LISTING 4.7: Service Façade sample implementation.

```
public class PaymentFacade : Façade  // Façade is abstract
{
private ProductManager m_oProduct = null;

    public PaymentFacade(){;}
public PaymentFacade(DataSet RawPacket) : base (RawPacket){;}
    public PaymentFacade(Packet oPacket) : base (oPacket){;}

    public override DataSet Execute(DataSet dsPacket, bool bCache)
    {
          Packet oPacket;

          // builds the packet from the raw dataset
          PreparePacket(dsPacket);

          if (Product == null)
{
              Product = CreateProduct(GetPacket());if (bCache)
              {
                  Product.PrepareCache();
              }
```

```
          }
          else{
             Product.Packet = GetPacket();}
          oPacket = Product.DoOp();

          // return raw packet back to caller
return PreparePacket(oPacket);  // this returns a DataSet
    // from a Packet using the
    // PacketTranslator
          }

          public ProductManager Product
          {
             get { return m_oProduct; }
             set
             {
                m_oProduct = value;
             }
           }

           public ProductManager CreateProduct(Packet oPacket)
           {
              ProductManager oProdMan;

    // packet type should have been set during PreparePacket()
    // in calling DoOp...
           switch (oPacket.Type)
           {
              case Constants.CREDIT_CARD_AUTH:
oProdMan = (ProductManager) new Product1(oPacket);
                 break;
              case Constants.CREDIT_CARD_SETTLE:
                 oProdMan = (ProductManager) new Product2(oPacket);
                 break;
              default:
                 oProdMan = null;
                 break;
          }
              return oProdMan;
   }

   // for testing only..
   public object SomeOtherBusinessFunction()
   {
         //…
   }
}
```

The Service Façade, once you take away some of the mentioned design options, is really a container with publicly accessible business methods. This example uses a single point of entry into the façade, as was demonstrated in the Service Factory sections earlier in this chapter. This was done to allow the Service Façade not only to be called from within a Web method but also to be used in the factory. This implementation, once plugged into a factory, delegates all specific business details to a Product Manager class (also described in this chapter). The PaymentFacade below is a ServiceFacade in charge of all credit card payment transactions. It can be called by several different Web methods (e.g., CreditCardAuthorize, CreditCardSettlement). Although most of the specific business rules are delegated to other classes, this façade understands one service type—payments. In essence, it is the kernel of the payment system. Using the data from the passed DataSet (e.g., Packet.Type), it will determine which Product class (e.g., CreateProduct()) should handle the incoming transaction. In our example, this is also where the packet is "prepared" and transformed into a more malleable data format, one that the business components of this type can easily work with. As you can probably surmise, this is only the tip of the iceberg. Much more functionality can now be place within the Service Façade. For the PaymentFacade, it would obviously be those features specific to payment processing. The point is that the ServiceFacade is the place to focus any high-level business design. For prototyping reasons, this may also be the place where you begin your conceptual work. It can become the best place to begin "sketching out" a high-level design.

Related Patterns

- Façade (GoF)
- Proxy (GoF)

ABSTRACT PACKET PATTERN

Intent

Provide an abstract container used to pass parameters to any objects within a framework. This will also serve to package discrete parameters into a single and more efficient data-marshaling package.

Problem

Whether you are working with Web services or any publicly available business function, eliminating unnecessary data traffic is important. Most server-based services will take a variety of parameters to perform any one business function. Variables anywhere from strings to bytes to arrays will need to be passed to these business services, and they should be passed in the most efficient manner. Most object-oriented designs call for some form of encapsulation. This may be data objects providing "accessors" (getters) and "mutators" (setters) used to interact with the business data. Interaction with these objects occurs through "hydration" using mutators or through "extraction" using accessors. This type of multiple round trip get/set interaction is fine when an object is local or when the interaction is simple. Multiple gets and sets across the network would not be good. In general, where this scenario falls short is when the objects involved are separated by some form of boundary.

Boundaries come in many forms. There are network boundaries, process boundaries, domain boundaries (.NET), and storage boundaries (I/O), etc. The problem is that as a developer, interacting with objects using several round trips to set and get data can become a problem—a performance problem. This is especially apparent when calling business objects across network boundaries where each get or set call adds a significant performance hit to the overall transaction. Aside from being wasteful in network usage, it also forces the server object to maintain state in between accessor/mutator invocations. In some cases, holding state may be necessary but this should be used carefully.

An option in avoiding multiple round trips during object interaction is to pass all parameters into the method at once. For some designs, this is perfectly fine. In those cases, the parameter list may include only one or two data elements that you must pass to an object. Most of time, however, this is not sufficient. One or two parameters can quickly become three, four, five, and many more. Maintaining business methods with long parameter lists (although done) is not recommended. This is where the Abstract Packet comes into play. It is simply a container for those parameters. It is a generic container with the ability to hold as many parameters as are necessary to facilitate any business object, as long as that business can receive that packet's data type (Figure 4.9). This also simplifies the signature of most business methods because now a business method can be typed with a single parameter. This also applies to return values. The return value can be of the same

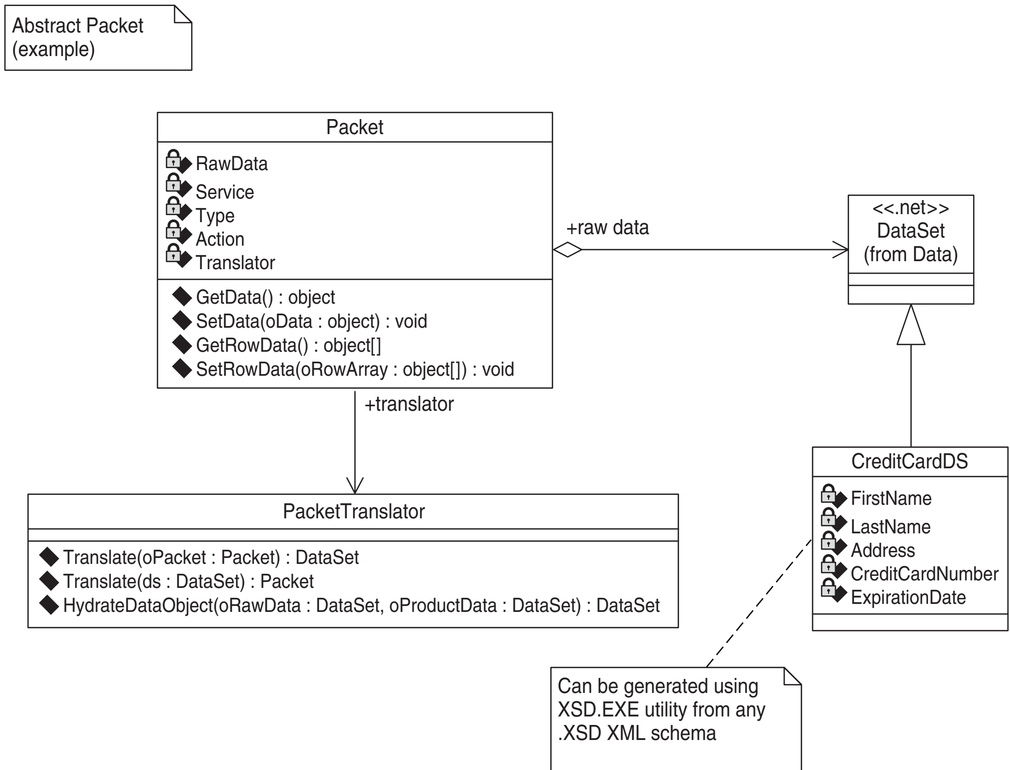


FIGURE 4.9: Abstract Packet implementation class diagram.

type, as long as that type is generic enough to contain data that will be returned from any business function.

Forces

Use the Abstract Packet pattern when:

- Web services will be used that contain more than two or three parameters.
- Business functions need to contain a standard signature contract to isolate future changes.
- Parameter types change frequently for business methods.
- Working with services crossing expensive boundaries (process, network, etc.).

Structure

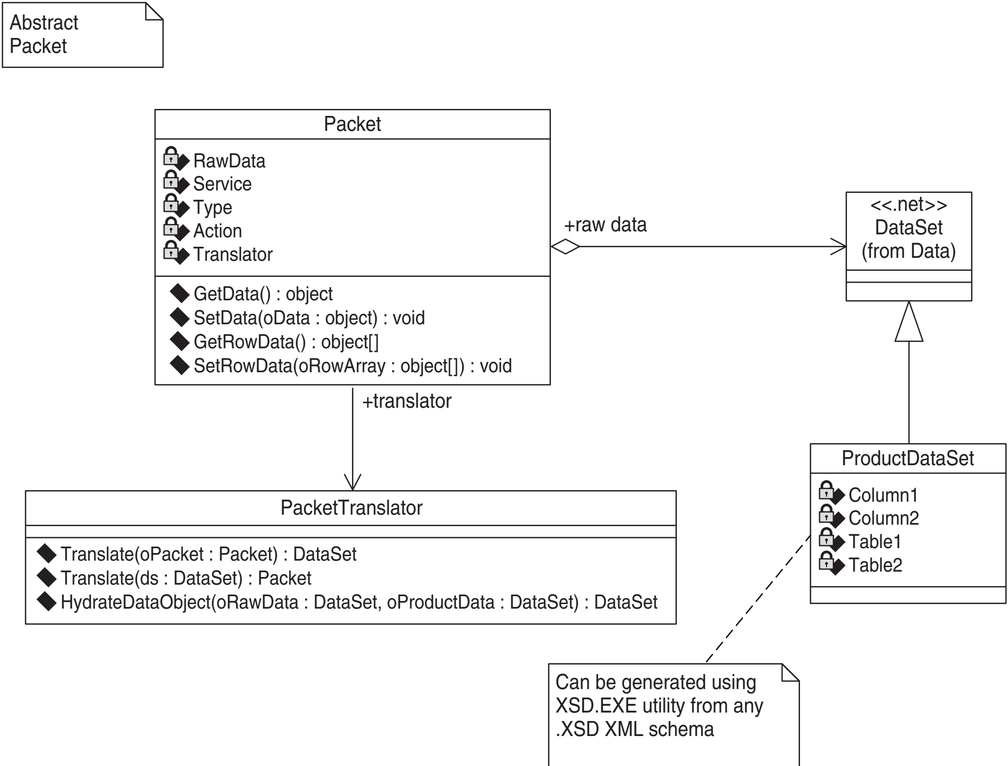


FIGURE 4.10: Abstract Packet generic class diagram.

Consequences

1. *Improves the parameter-passing efficiency.* When all parameters are bundled into one object, the developer will have more control on how to marshal those parameters. This includes any serialization that may take place. This also provides a controlled means of retrieving or unpackaging those parameters within the containing object.

2. *Provides a container in which to build dynamic parameter sets.* For those scenarios where the set of parameters can vary frequently, the Abstract Packet provides the container in which to build such a facility. In our example, the Packet class simply aggregates an already dynamic data type in the form of an

ADO.NET DataSet object. Using the DataSet data member, the Packet can take any shape and contain any data. As long as the business method that receives this packet knows how to interact with it, the packet can be used throughout the framework. The packet then can contain any data and be passed to each tier of the framework without implementing specific behavior for each representation of the packet. Only the business method that must directly interact with the packet's data must know what data elements it requires. For example, at this stage, the business method would call the packet's GetData() and request specific fields from the packet. The packet, in turn, delegates the lookup to the aggregated DataSet. To the rest of the system, this is just a generic packet.

Look Ahead

Another option to this pattern is to bind a "type-strong" Data Service object that will be a child of a DataSet and, thus, can also be bound to the packet when the packet is built or translated. This new option provides a type-strong DataSet that any business method wishing to interact with the packet can use instead of using the packet's delegation methods. Using a type-strong DataSet is one way to avoid boxing/unboxing and can improve performance. Not to mention that it provides a much friendlier development environment for Visual Studio .NET users, especially those who love Intellisense. Using a type-strong DataSet will be fully discussed in the Chapter 5.

3. *Eliminates the binding of business methods to technology-specific data types, such as those in ADO.NET (DataSet).* This simply avoids forcing business methods from including ADO.NET types in their signatures and provides another level of abstraction.

4. *Hides the implementation details of the aggregated inner type (a DataSet, in this case).* Business methods, even those directly interacting with the data, do not require any of the details for manipulating types such as a DataSet. Those services directly interacting with the packet can simply use the methods provided by the packet. Methods such as GetData() require only parameters such as a field name to retrieve the underlying information. Keep in mind that a DataSet does not have to be bound to an actual database; a field name can be just a name of a column from the DataSet that could have been generated dynami-

cally. As mentioned earlier in the *Look Ahead* sidebar, there is also another means of interaction (see Chapter 5).

Participants

- Packet (Same name as implementation)—This is the Abstract Packet itself. This class acts as a form of "flyweight" in that its job is to contain data that can be shared efficiently with the rest of the system. It will act as a container of both extrinsic (passed-in) and intrinsic (static) data used by the rest of the system to route a request and perform an action.
- Packet Translator (same)—This includes any packet translation that constructs a packet using an overloaded method called *Translate()*. The Translator constructs the destination object and maps the appropriate values into the new data object. This construction and translation logic is business-specific. The goal is to simplify and abstract this logic. The client does not know or care how the construction or translation takes place or which Translate method to call. The client simply invokes Translate(), and the overloaded method takes care of invoking the appropriate method based on the type passed. Refer to the Packet Translator section later in this chapter for details.
- DataSet (same)—This is a standard ADO.NET DataSet object. This can represent any data schema, whether it is based on a persistent data model or not. This becomes the actual data container with a callable wrapper. This wrapper is the Packet class. The packet holds descriptive data elements to identify the packet, which can then be used for routing or other logic. Any other generic container such as an ArrayList, object[], etc., can be used as well.
- ProductDataSet (CreditCardDS)—This is the business-specific data services class (data access object). This is a classic data services object that directly represents a view on a database or other persistent set. This class is strongly typed to the specific data elements of a particular business service or database. It inherits from a DataSet to gain any ADO.NET features, such as serialization and XML support, to allow it initialize or extract the data once it is hydrated.

Implementation

The Abstract Packet was implemented primarily to aggregate a DataSet. In fact, the DataSet type in .NET can be used as an Abstract Packet with and of itself. For a

technology backgrounder on ADO.NET and DataSets in particular, please refer to Chapter 5. Those already familiar with DataSets will understand that a DataSet is a generic object that can hold just about any data representation in memory. A DataSet can be dynamically built and hydrated from a database or, as is the case in this example, be hydrated from an XSD schema. The beauty of our Abstract Packet example is the fact that it does not "reinvent the wheel." The Packet class does not try to duplicate functionality that a DataSet already provides. It simply delegates to it and acts as an aggregator of an existing Dataset. The other data members of the Packet class are simply used to identify the packet for use by the architecture as this packet gets passed from service to service.

To build a packet, one must first have a DataSet object. In our example, a Web service receives and returns the DataSet type. When a DataSet is passed into our Web service, it instantiates and builds an appropriate Packet object. The building step can become complex and, therefore, should be farmed out to another service, such as a Packet Translator (covered later in this book). The primary step in building a packet is simply to set the DataSet as a data member of the packet. This is done using the Data property of the packet. The remaining properties of the packet are optional. More properties can be added to the packet as needed by the business requirements. The point is that the DataSet, now a member of the packet, still contains most of the data. When data needs to be extracted from a packet, its GetData methods or indexers can then be called, which delegates to the DataSet. The Packet class can now become the primary parameter passed to all business methods. This is similar to the functionality of an Adapter Pattern (GoF).

A DataSet could have been passed instead, but using a Packet class provides another level of abstraction. This abstraction will safeguard those methods from change and provide a high-level interface to those services that may not need to know how to manipulate a DataSet directly. The DataSet can be as simple as representing a single table or as complex as representing an entire database with constraints and all. By using a DataSet, all data can be treated as though directly contained within an actual database. This is true even if the DataSet is strictly represented in memory. Within the Packet class, methods can be designed to manipulate the DataSet in any way it sees fit. One caveat to this particular implementation, however, is the fact that the Packet class does not contain any type-specific methods. For example, each overloaded SetData() method takes an object as one of its parameters. Although this facilitates setting any data type of any field in the DataSet, this also introduces what .NET refers to as *boxing*. It is

recommended that for performance-intensive implementations, type-specific methods should be created to avoid this side effect.

Technology Backgrounder—Boxing/Unboxing

Those already familiar with details behind value types, reference types, and the process of boxing/unboxing can skip this section. For those wanting more information, read on.

In the .NET CLR, you have two general types: value types and reference types. Value and reference are similar in that they both are objects. In fact, everything in the CLR is an object. Even value types are objects in that they have the System.ValueType as a parent class, which has System.Object as its parent. Each primitive type is represented by an equivalent class. For example, the primitive types of int and long in C# both alias the System.Int32 and System.Int64 classes, respectively, both of which have System.ValueType as parent. Other value types include structs and enumerations (enums). If it inherits from System.ValueType, it is treated as a value type in the CLR.

Value types are handled a bit differently than reference types in that they are passed by value. Passing by value means that a copy of the value is made prior to calling the function. For most value types, the cost of making this copy is small and usually outweighs the performance issues that arise when dealing with reference types. Value types represent a value that is allocated on the stack. They are never *null* and must contain data. Any custom value type can be created simply by deriving from System.ValueType. When creating your own value types, however, keep in mind that a value type is *sealed*, meaning that no one else can derive from your new type.

Reference types are based on the heap and can contain null values. They include types such as classes, interfaces, and pointers. These types are passed by reference, meaning that when passed, the address of the object (or pointer) is passed into the function. No copy is made. Unlike value types, when you make a change, the original value is changed, as well, because you are now dealing with a pointer. Reference types can be used when output parameters are required or when a type consumes a significant chunk of memory (remember that structs are value types, and they can grow quite large). However, they also must be managed by the CLR. This means that they must be kept track of and garbage collected. This also will add a performance penalty. Value types should be used wherever possible to improve performance and to conserve memory. If your object consumes a lot of

memory, a reference type should be used, bearing in mind that any destruction or finalization of your type is going to be nondeterministic.

Once you understand the technical differences of how value types and reference types are treated, you will understand how *unboxing* and *boxing* work. Values types can become reference types, and the opposite is true as well. This can be forced or this can be automatic. The CLR will automatically convert a value type into a reference type whenever needed. This is called *boxing*. Boxing refers to converting a stack-allocated value into a heap-based reference type. An example of this would the following:

```
int nFoo = 1;// nFoo is a value type
object oBar = nFoo;     // oBar is a reference type of type
// System.Object
```

Here, a box is created and the value of nFoo is copied into it. To translate, heap space is allocated, and the value of nFoo is copied into that memory space and now must be temporarily managed. When a value is boxed, you receive an object upon which methods can be called, just like any other System.Object type (e.g., ToString(), Equals(), etc.). The reverse of this process is called *unboxing*, which is the just the opposite. A heap-based object is converted into its equivalent stack-based value type, such as:

```
int nFoo = (int)oBar;// oBar is a reference type
```

Unboxing and boxing, although convenient, can also become a small performance bottleneck and should be used with care. For methods that will be called extremely often, as will our Packet data object, using a System.Object type as a parameter where value types will be expected should anticipate a low performance. This is due to boxing. Methods such as these can be changed to support a System.ValueType but you must also create methods to except other types, including strings (which, by the way, are not value types).

Most of the methods and indexers defined in this class delegate to the Data property. The Data property simply returns the m_dsRawData member variable of this class (which is the DataSet we are wrapping). The Packet class uses this property to delegate most of the calls to the wrapped DataSet to return data, set data, and so on. This uses the DataSet for the heavy lifting. Wrapping the DataSet in this aspect gives the Abstract Packet its “Adapter” qualities, allowing it to be

passed to all business services that accept a Packet data type. Listing 4.8 contains code for a typical Abstract Packet implementation.

LISTING 4.8: Typical Abstract Packet implementation.

```
public class Packet : IDisposable
{
      private DataTableCollection m_oData = null;
      private DataTable m_dtMeta = null;

      private DataSet m_dsRawData = null;

      private string m_sType;
      private string m_sService;
      private string m_sAction;

      private PacketTranslator m_oTranslator = null;

      public Packet()
      {
         RawData = new DataSet();
         Translator = new PacketTranslator();
      }

      public Packet(PacketTranslator oTranslator) : this()
      {
         m_oTranslator = oTranslator;
      }

      public static bool operator == (Packet p1, Packet p2)
      {
         if ((object)p1 == null && (object)p2 == null)
            return true;
         if ((object)p1 != null && (object)p2 == null)
            return false;
         else
            return p1.Equals((Packet)p2);
      }

      public static bool operator != (Packet p1, Packet p2)
      {
         if ((object)p1 == null && (object)p2 == null)
            return false;
         if ((object)p1 != null && (object)p2 == null)
            return true;
         else
            return !p1.Equals((Packet)p2);
      }
```

```
public DataSet RawData
{
   get { return m_dsRawData; }
   set { m_dsRawData = value; }
}

public DataTableCollection Data
{
   get { return m_oData; }
   set { m_oData = value; }
}

public DataTable Meta
{
   get { return m_dtMeta; }
   set { m_dtMeta = value; }
}

public string Type
{
   get { return m_sType; }
   set { m_sType = value; }
}

public string Action
{
   get { return m_sAction; }
   set { m_sAction = value; }
}

public string Service
{
   get { return m_sService; }
   set { m_sService = value; }
}

public PMPacketTranslator Translator
{
   get { return m_oTranslator; }
   set { m_oTranslator = value; }
}

public string TransId
{
   get { return m_sTransId; }
   set { m_sTransId = value; }
}

// assume first table in collection
public object GetData(string sColumn)
```

```
         {
            DataRow dr = null;
            object oReturn = null;

            if (Data[0].Rows.Count > 0)
            {
               dr = Data[0].Rows[0];
               oReturn = dr[sColumn];
               if (oReturn == System.DBNull.Value)
                  oReturn = null;
            }
            return oReturn;
         }

         public object GetData(string sTable, string sCol)
         {
            return GetData(sTable, sCol, 0);
         }

         public object GetData(string sTable, string sCol, int nRow)
         {
            DataRow dr = null;
            object oReturn = null;

            if (Data[sTable].Rows.Count > 0)
            {
               dr = Data[sTable].Rows[nRow];
               oReturn = dr[sCol];

               if (oReturn == System.DBNull.Value)
                  oReturn = null;
            }
            return oReturn;
         }

         public object[] GetRowData(string sTable, int nRow)
         {
            object[] oRowArray = null;

            if (Data[sTable].Rows.Count > 0)
            {
            oRowArray = Data[sTable].Rows[nRow].ItemArray;
            }

            return oRowArray;
         }

public void SetRow(string sTable, int nRow, object[] oaRow)
         {
            Data[sTable].Rows[nRow].ItemArray = oaRow;
```

```
          }

          public void SetData(string sCol, object oVal)
          {
             DataRow dr = null;

             if (Data[0].Rows.Count > 0)
             {
             dr = Data[0].Rows[0];
             dr[sCol] = oVal;
             }
          }

public void SetData(string sTable, string sCol,
    object oVal)
          {
             SetData(sTable, sCol, 0, oVal);
          }

public void SetData(string sTable, string sCol, int nRow,
    object oValue)
          {
             DataRow dr = null;

             if (Data[sTable].Rows.Count > 0)
             {
                dr = Data[sTable].Rows[nRow];
                dr[sColumn] = oValue;
             }
          }

          public string this[string sColumn]
          {
             get
             {
                return Convert.ToString(GetData(sColumn));
             }
             set
             {
                SetData(sColumn, value);
             }
          }

          public string this[string sTable, string sColumn]
          {
             get
             {
return Convert.ToString(GetData(sTable, sColumn));
             }
             set
```

```
            {
                SetData(sTable, sColumn, value);
            }
        }

        public string this[string sTable, string sColumn, int nRow]
        {
            get
            {
                return Convert.ToString(GetData(sTable, sColumn, nRow));
            }
            set
            {
                SetData(sTable, sColumn, nRow, value);
            }
        }

        public static string SafeCastString(object oValue)
        {
            string sReturn;

            if (oValue != null)
                sReturn = (string)oValue;
            else
                sReturn = "";

            return sReturn;
        }

        public static decimal SafeCastDecimal(object oValue)
        {
            decimal dReturn;

            if (oValue != null)
                dReturn = (decimal)oValue;
            else
                dReturn = (decimal)0.0;

            return dReturn;
        }

        …

        public void Dispose()
        {
            RawData.Dispose();
        }

    }
}
```

Related Patterns

- Value Object (Alur, Crupi, Malks)
- Adapter (GoF)
- Composite (GoF)

PACKET TRANSLATOR

Intent

Facilitate the construction or mutation of complex data elements or objects into separate forms without knowing the details of the objects being translated. Translate Abstract Packets from one representation to another.

Problem

Similar to that of the "gang of four" Builder pattern, the Packet Translator separates the construction of a complex object from its representation. However, in addition to the Builder, this also facilitates a method to translate the complex object from one type to another and back again. The point of this translation service is to provide a means upon which to place the logic necessary to receive data packets of one format and convert them into another format. This is extremely typical when implementing an Abstract Packet pattern or any general object containing data parameters that must pass into another section of code that may understand a different set of values. For example, the Abstract Packet object a designer uses to pass data into the system may be quite different than the object used to pass from business service to business service. Different layers of the system or even different tiers sometimes require different parameter "packaging" rules. The data elements necessary for parameter passing at a business tier may be completely different for what is required at the persistence or data tier.

What this pattern solves is a fixed method of translation such that the implementation of the translation is generic and separated from its representation. Like an Adapter Pattern (GoF), this pattern provides an object different in interface yet containing similar internal data. The Packet Translator not only centralizes the translation implementation of turning one packet format into another but also provides a standard set of methods to do so. The calling of a Packet

Translator will typically occur in a base class hidden from any concrete client, such as a Service Façade.

Forces

Use the Packet Translator Pattern when:

- Using an Abstract Packet in your framework.
- The format of received data does not match that used within the framework.
- You want to isolate complex construction logic for an Abstract Packet.
- You want to standardize the packet-building process within a framework.

Structure

FIGURE 4.11: Packet Translator generic class diagram.

Consequences

1. *Encapsulates complex construction details of an Abstract Packet or any complex object.* Typically when working with more than one set of data parameters, the logic used to map those parameters can become complex. This is especially true when incoming parameters do not directly match that of outgoing parameters. In our example, a DataSet is the external object format that must be translated into another Abstract Packet format called *Packet*. The construction of the Packet class can become complex and should be delegated to another entity. Complex construction such as this should also be abstracted in the likelihood that other translations may occur using different object formats.

2. *Provides a standard means of translating two object formats into one another.* Translating packets can soon become a systemwide process. If there is more than one public entry point into an existing framework, this pattern will provide a standard means by which to translate packets of any format.

3. *Centralizes the construction and binding of type-strong objects into an Abstract Packet.* Although this could be considered an optional feature of the Packet Translator, creating type-strong data objects is a preferred approach when binding the data that will either reside on or be aggregated by the Abstract Packet. HydrateDataObject() in the Packet Translator can also be used from a factory to create the appropriate type-strong data object that must be bound to an Abstract Packet. Please refer to the Abstract Packet Pattern section earlier in this chapter for more information. The implementation section below will explain how our example utilizes a type-strong data object and binds it to an Abstract Packet, using the Packet Translator.

Participants

- ConcreteFacade (CreditCardFacade)—This is simply the driver object of the pattern. This entity can be any client that directly interacts with the Packet Translator. In our example, this is a CreditCardFacade object that during construction receives an external packet and translates it into a Packet object. Actually, in the CreditCard production application, the logic for packet translation lies in the parent class of all façades, alleviating the need to duplicate this process in each ConcreteFacade implementation. This is not a requirement, however.

- PacketTranslator (Same name as implementation)—This is the heart of the pattern. All pattern logic is implemented here. This includes translation that constructs both object formats using an overloaded method called *Translate()*. The ConcreteFacade simply has to call Translate() on the Packet Translator, passing in the appropriate data object. In our example, if a business service is called, passing an external data object such as a DataSet, Translate () is then called, passing the DataSet to the Translator. The Translator constructs the destination object and maps the appropriate values into the new data object. This construction and translation logic is business-specific. The goal is to simplify and abstract this logic. The client does not know or care how the construction or translation takes place or which Translate method to call. The ConcreteFacade just invokes Translate(), and the overloaded method takes care of invoking the appropriate method, based on the type passed. Keep in mind that if you are using data objects of the same data type, an overloaded Translate method will not work because the signature will be the same. For those cases, a different Translate method should be created, such as TranslateA() and TranslateB(). The remaining piece of translation is the optional use of a type-strong data object in the translator. This is not a requirement but when using an Abstract Packet that does not include typed methods, this can improve performance of your application. The first step to implementing either Translate() method is to construct the destination data object. For those cases where a type-strong version of that data object can be used, a factory method should then be implemented to perform this construction. The factory method will construct a type-strong class based on some information in the received data object. For example, when receiving a DataSet type data object, HydrateDataObject() is called from the initial Translate method. In fact, this Translate method becomes our factory (see below). Here, the correct type-strong class is instantiated, hydrated with the incoming data from the DataSet, and returned to the calling Translate() method to complete the translation. Naming this method *HydrateDataObject()* seemed appropriate because an XML schema with instance data was used to "hydrate" our type-strong data object once it was constructed. How the data is actually "sucked" into in the newly constructed data object is up to the developer, and again, this is optional for the pattern. For more information on type-strong data objects, please refer to Chapter 5.

- Packet (same)—This is the destination data object. This is typically an Abstract Packet that will act as a container to all business data passed to each method in the framework. The business methods in the framework "speak" the Packet language while the external world speaks the "DataSet" language. DataSets are mentioned next.

- DataSet (same)—This represents the external data object. This type is passed into our business framework from the outside world. A DataSet is a great choice for this data object, due to its dynamic nature, flexible representation, and the fact that many .NET tools down the line will be supporting it. This type is optional. Other types could have been chosen for the PacketTranslator pattern, and for smaller applications, a DataSet could be considered overkill. For simpler cases, an ArrayList could have been used or even a custom data object. Keep in mind, however, that choosing a custom data object brings with it unique challenges to the data-marshaling world, and it is recommended that you stick with a "standard" .NET data type. This is especially true when using SOAP as your transport protocol.

- ProductDataSet (CreditCardDS)—This represents the type-strong data object. This inherits from the DataSet and again is optional. Type-strong data objects are discussed in Chapter 5.

Implementation

Figure 4.12 looks much more complicated than it really is. The base of the pattern lies in the encapsulation of the packet construction. All construction and translation take place in one location—PacketTranslator. Where the class model becomes more complex is when a type-strong data object is used. In our case, that type-strong data object is a child class of a DataSet called *CreditCardDS*, and using it (as mentioned many times in this section) will be one of the focuses of Chapter 5.

The code in Listing 4.9 is implemented in the client of the translator. In our example, this is the CreditCardFacade class. It simply takes an external DataSet object, instantiates the PacketTranslator class, and calls Translate. It, like the Translator, uses an overloaded method called *PreparePacket* to alleviate its client from having to know which method to call. The return value of each PreparePacket is the appropriately formatted data object.

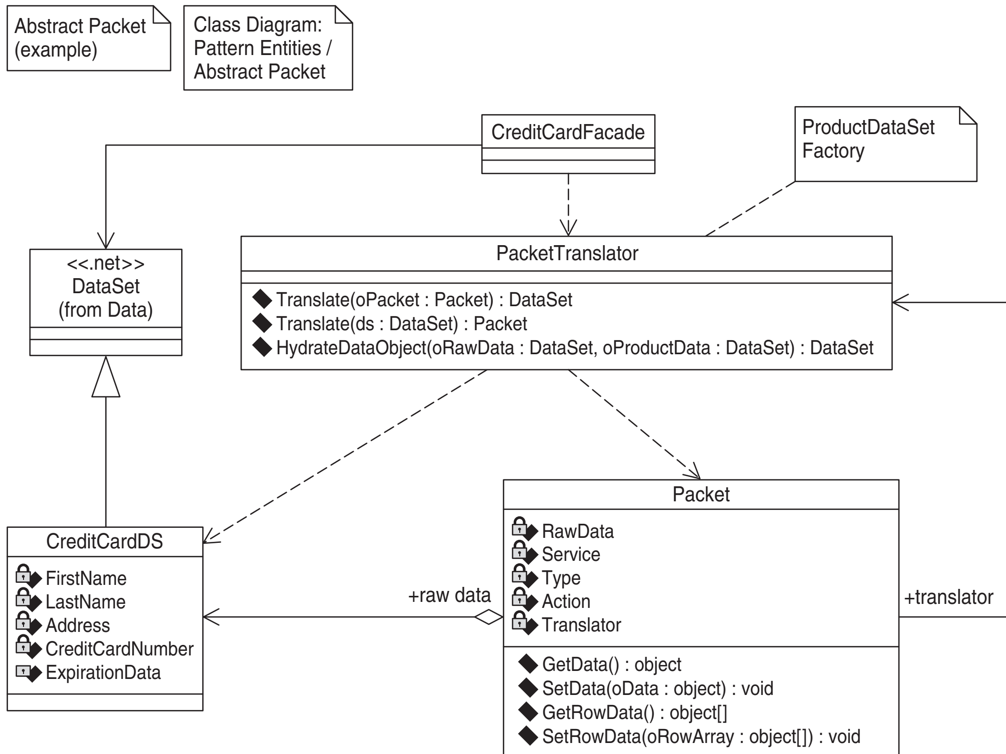


FIGURE 4.12: Packet Translator implementation class diagram.

LISTING 4.9: Abstract Packet sample implementation—preparing packets.

```
public Packet PreparePacket(DataSet dsRawPacket)
{
try
    {
            SetRawPacket(dsRawPacket);
PacketTranslator oPacketTranslator = new
PacketTranslator();

SetPacket(oPacketTranslator.Translate(
GetRawPacket()));
    }
catch(Exception e)
    {
    …
    }
    return GetPacket();
```

```
}

public DataSet PreparePacket(Packet oPacket)
{
    try
    {
          SetPacket(oPacket);
          PacketTranslator oPacketTranslator = new
    PacketTranslator();

    SetRawPacket(oPacketTranslator.Translate(
    GetPacket()));
    …
    return GetRawPacket();
}
```

The code in Listing 4.10 shows the implementation of each Translate method in the PacketTranslator object, along with our HydrateDataObject(). In this example, a DataSet is received, and in the above PreparePacket, the Translate(dsRawPacket) is called. Here the Translate method acts as factory and instantiates the appropriate type-strong data object. Because each type-strong data object inherits from DataSet, the returned type from HydrateDataObject is of type DataSet. In fact, as was mentioned earlier, the Packet type simply contains our DataSet. For those cases that use type-strong data types, this type can actually be downcast to the appropriate type-strong DataSet and later accessed by those methods wishing to interact with type-specific behavior. This is great for Visual Studio's Intellisense! To construct our destination Packet, HydrateDataObject() is called, passing into it both the incoming DataSet via dsRawPacket and the newly instantiated type-strong DataSet called *CreditCardDS*. Here in HydrateDataObject(), we use the XML serialization services of .NET to perform the data hydration of the destination object (see the technology backgrounder in Chapter 5). Once hydrated, it is returned to Translate(), which in turn returns the entire packet back to PreparePacket(). You should also notice that before HydrateDataObject() is called, members of the Packet are filled with high-level data that will be used to route this packet. This is optional but points out that this is the place to implement such construction behavior. Finally, the other Translate method that is called with a packet must be turned into a DataSet. This is much simpler, at least in our example, because the DataSet is already contained in our Packet class on the way out and simply needs to be returned as is.

LISTING 4.10: Packet Translator sample implementation—translating packets.

```
private DataSet HydrateDataObject(DataSet dsRawPacket,
                                                DataSet oDataObject)
    {
    System.IO.MemoryStream stream = new
    System.IO.MemoryStream();
          dsRawPacket.WriteXml(new XmlTextWriter(stream, null));
          stream.Position = 0;
          oDataObject.ReadXml(new
    XmlTextReader(stream),XmlReadMode.IgnoreSchema);
          return oDataObject;
    }

    public Packet Translate(DataSet dsRawPacket)
    {
          Packet oPacket = new Packet(this);

          // fill in packet values from DataSet (rawPacket)
          oPacket.Type = GetType(dsRawPacket);
          oPacket.Service = GetService(dsRawPacket);
          oPacket.Action = GetAction(dsRawPacket);

          switch (oPacket.Type)
          {
             case (Constants.CREDIT_CARD_TYPE):
             {
    oPacket.RawData =
    HydrateDataObject(dsRawPacket, new
    CreditCardDS());
                break;
             }
             case (Constants.TYPEB):
                {
                   …
                   break;
                }
             default:
             {
                oPacket.RawData = dsRawPacket;
                break;
             }
          }

          return oPacket;
}

public DataSet Translate(Packet oPacket)
```

```
{
    return oPacket.RawData;
}
```

Related Patterns

- Value Object (Alur, Crupi, Malks)
- Builder (GoF)
- Abstract Packet (Thilmany)