New information has been added to this article since publication.
Refer to the Editor's Update below.

INSTRUMENTATION

Powerful Instrumentation Options in .NET Let You Build Manageable Apps with Confidence

Jon Fancey

This article discusses:

- Instrumentation in the managed world
- What instrumentation can help you measure
- Performance counters, WMI, tracing, and other measuring techniques
- Choosing the best instrumentation for your purposes
- Security considerations

This article uses the following technologies:
WMI, .NET, Visual Basic .NET

Instrumentation allows you to determine the running state of a system once it has been deployed. This is crucially important today since the people supporting systems are unlikely to be the same people who developed them. Like good error handling, instrumentation is best done at development time. A sound instrumentation policy must be established at the beginning of the development process to determine what should be instrumented, why, where, and how.

Instrumentation can be performed using a wide variety of available technologies that provide high-quality support and diagnostic information from within an application itself. The technology I will focus on here is a mixture of core traditional Windows® facilities—event logging, debug tracing, performance counters, and Windows Management Instrumentation (WMI). I will also cover the Enterprise Instrumentation Framework (EIF) and features provided by the Microsoft® .NET Framework such as tracing and structured exception handling (SEH). The .NET Framework minimizes the overhead and makes all of this easy to use, which is a key point since business users often consider instrumentation to be a nonessential process.

The Landscape

Distributed applications are getting more and more complex. When you start looking at systems in their entirety, the complexity can be daunting. I'm going to focus on the challenges faced by enterprise designers and developers who are building complex distributed apps. Whether you're building Web sites, services, or traditional client-server applications, you'll find the task of managing these applications to be quite a challenge due to the huge invisible architectures.

Instrumentation can help by allowing you to answer questions such as: what went wrong where and how often does that happen? How do I know if an application is alive? How many business transactions are performed hourly? What's the maximum throughput that can be handled? What's the busiest time for various activities? How do the applications scale up and out? How do I know who's doing (or trying to do) what?

These questions fall into four distinct categories: performance, reliability, scalability, and security. Performance is crucial to support service-level agreements (SLAs) and reliability for Quality of Service (QoS) contracts, especially with third parties. Scalability is important for growth, including sales and marketing opportunities. Security, of course, is always critical.

One of the first things you must recognize is that errors should be anticipated. Deciding how to handle them is not a runtime-only decision, but a design decision. Leaving it until the coding stage will often lead to a weak and incoherent solution that does little to provide good instrumentation.

What to Measure and How to Plan

After deciding to instrument while you're writing your application logic, the second most important thing is to decide exactly what needs instrumenting. User details, date/time, and thread/process information (crucial for providing sequencing) are absolute essentials. You will also want to include your application name and where the event was logged (class/method names). Other common useful information includes state information, such as parameter values and current context values. You should also be careful to avoid calling information that is of little use, such as logging disk I/O activity, total system threads, and so on. You should include the essentials plus anything specific to the task your code is performing, such as database connection details (but be careful since exposing sensitive data in this way can create a security risk).

You must decide what the monitoring and support requirements are for your project and document them. This task involves the development team as well as members of your testing and support teams. You can then determine where to include instrumentation on the requirements that are determined by these three groups. You have to take a holistic approach so that instrumentation is present throughout your design. Any missed areas will be excluded from your monitoring and can adversely affect your project.

New Possibilities with .NET

If you're coming from a Visual Basic® 6.0 background, the .NET Framework probably looks like an oasis in the desert. The challenge used to be getting all the core functional requirements into an application. Instrumentation was impossible to do well without a great deal of time and effort. The solutions tended to be either too complex (tricky C++ DLLs) or overly simplistic (App.LogEvent). It was easy to introduce an unacceptable performance overhead which often led to incomplete instrumentation.

The irony of this is that you can't really tell how fast parts of an application run without instrumentation. Capacity planning (the analysis of anticipated load versus actual capability) is a key activity that is still relatively marginalized when it comes to Web-based distributed applications. It tends to be performed on the surface of the application, typically from virtual clients such as Mercury Interactive's LoadRunner or the Microsoft ACT tool. If the app starts getting slow, where is it slow? Is everything slow? That's highly unlikely, but you need to be able to prove it. Good instrumentation can answer questions like this and remove much of the guesswork.

Software development is going through a fundamental change, and developers are still trying to catch up with it. The typical development approach to Web-based applications is to divide up tasks for individual developers to build a small part of a new system and test it on individual machines. They then deploy it to a testing environment along with the rest of the team's work. This process is fundamentally flawed. Web-based applications are developed and tested on single workstations where everything performs fine. Often no regard is given to the fact that thousands of users could be hitting the application once it's live, rather than just a single user. Build verification testing can help a great deal to address this problem by prescribing early integration.

The bottom line is that complex distributed applications are unmanageable without good instrumentation. So let's take a look at what you can accomplish with instrumentation. First I'll review the traditional options and then examine the new possibilities that are available in the .NET Framework.

Event Log

The event log (eventvwr.exe) has always been part of Windows NT®. It provides a central place to log information and is divided into three main categories: application, system, and security. You can write to any of these categories, but usually you should stick to the application group since this is the most relevant. It's as easy as executing the following code in Visual Basic .NET:

```
Imports System.Diagnostics

EventLog.WriteEntry("MyAppSource",
                    "Details",
                    System.Diagnostics.EventLogEntryType.Information)
```

There are a couple of things to point out here. If you've used the Visual Basic 6.0 App.Logevent, you'll notice that you can define your own event sources at will (no long list of VBRuntime entries in the event viewer anymore). Also, it used to be a real pain to create the descriptions for events because they involved building a message compiler resource file and linking it into your application DLLs. The .NET Framework now performs this task for you. Behind the scenes, the mechanism is the same. Look in the registry under the following key:

```
HKLM\System\CurrentControlSet\Services\EventLog\Application\MyAppSource
```

If you put that previous code snippet in a Windows Application project this key will be there. It is read by eventvwr.exe (or anything else) to locate the descriptions for the events you raise when they are displayed. This points to a generic event description handler provided by the .NET Framework located at:

```
%WINDOWS%\Microsoft.NET\Framework\%version%\EventLogMessages.dll
```

This is a resource-only DLL which has a load of replaceable %1 parameters to wire up the descriptions you specify in your code to the event viewer. The registry entry is added at run time when the statement is actually executed, not at compile time. You may find, though, that you are quickly building up junk in the registry under the Application key because each time you change the source argument, you'll add a new subkey. These subkeys will remain in the registry permanently unless you deliberately remove them (another reason to think about instrumentation early in development).

Performance Counters

Performance counters were difficult to implement in the days of Visual Basic 6.0. The performance counters API was impossible to use directly from Visual Basic 6.0 due to the entry points that need to be defined in your DLLs—everything in Visual Basic was emitted as COM CoClass with four standard DLL entry points and no way to add your own. The typical setup was to use a helper DLL that the performance monitor (perfmon.exe) would load and talk to in order to get your application statistics. Your app would be instrumented to make calls to update the counters at the appropriate points. This was nontrivial, as some kind of shared memory was required between object instances to maintain the counter values. This is because Perfmon runs in a separate process, calling a central collection function (residing in the Perfmon process) in your application that is independent of your individual objects. The .NET Framework makes these headaches go away. **Figure 1** shows the current performance counter architecture. The .NET Framework component, netfxperf.dll, runs in the Microsoft Management Console process mmc.exe that hosts the performance counter snap-in. This Process (B) loads the CLR and talks to your application in Process A to gather the statistics.

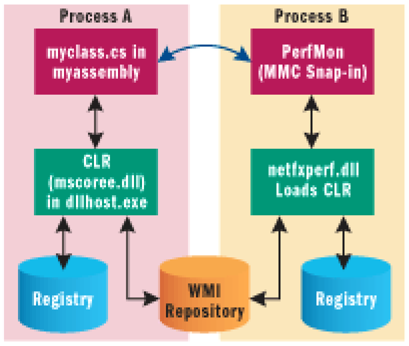

Figure 1 Performance Counter Architecture

In order for the Perfmon snap-in (or anything else) to find your counters, a number of registry settings must be added. The .NET Framework conforms to this automatically, as with event logging, when it hits the actual line of code. There is one main key that is created:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet    \Services\category\
```

This key usually contains a single subkey named Performance.

By using a generic DLL for this, the only issue to solve is how your assembly is associated with the key so that netfxperf.dll knows where to get your counters (and their values) when loaded by the snap-in. The solution uses the Wbem values that exist under this key. These values are used to obtain your counters using WMI. The .NET Framework registers your counters with the underlying WMI repository when it executes your code. Later, when Perfmon comes along, it retrieves your counter names from the registry, and using WMI, makes a request to be connected to them. If your application is running, WMI will provide the connection to the actual values. Incidentally, this is accomplished using the automatic discovery and purge feature of WMI, which is why some of the registry values are prepended with WbemAdap (WMI is an implementation of Web-Based Enterprise Management, also known as WBEM). The other registry counter values, First Counter/Help and Last Counter/Help, have to be strung together across all the counters on your system. In the past this could be accomplished by calling the lodctr/unlodctr API functions. If you get this wrong it can be a real problem because you will affect the ability of Perfmon to enumerate any of the counters on your machine. Again, the .NET Framework takes care of all of this for you.

The .NET Framework supports the full range of available Windows counter types. The most useful ones are the absolute, average, and timing groups. These allow you to represent such things as the number of SQL commands executed by your application, the average time it takes to compile a certain value, or the number of Web service requests per second.

Creating a performance counter and incrementing its value in .NET is as easy as the following:

```
Imports System.Diagnostics

Dim objCat as PerformanceCounterCategory
Dim objCtr as PerformanceCounter

objCat = PerformanceCounterCategory.Create("MyApp", _
                                           "MyApp help", _
                                           "counter 1", _
                                           "counter 1 help")

objCtr = new PerformanceCounter("MyApp","counter 1",false)
objCtr.Increment()
```

Again, you need to be careful that during development you don't make too much of a mess of your registry. When the category's Create method is hit, the MyApp key will appear under the registry path just shown. When the new PerformanceCounter is hit, the Wbem values are created during the WMI ADAP processing.

Tracing

The Win32® debugging API has always been a handy tool. The OutputDebugString function can be added to applications to provide dynamic tracing so that you can actually see an application's debugging information using a tool like DbgView.exe (http://www.sysinternals.com) or DBMON (provided with the Platform SDK). You can even connect to it from a remote machine. The .NET Framework still makes use of this API in its tracing implementation, which is what I'll discuss next.

Tracing is a useful idea. You want to see which methods are touched in what circumstances: ASP.NET provides the Trace method mechanism in the System.Web.TraceContext class while the base class library (BCL) provides the Trace and Debug classes in the System.Diagnostics namespace. To use System.Diagnostics tracing from your applications you must add tracing statements to your code and ensure that tracing is enabled at compile time. Tracing statements take one of several similar forms. The most commonly used form looks like this:

```
System.Diagnostic.Trace.WriteLine("something useful")
```

Both System.Diagnostics.Trace and System.Diagnostics.Debug are sealed classes. The main difference between them is the conditional compilation string used to include calls made on these classes in the build. The Debug class methods will only be compiled into your builds when the DEBUG compilation constant is set, whereas the Trace methods will be compiled into any build that has the TRACE compilation constant set (see the SDK documentation for full details). You can also set the tracing level in your application .config file so that you have some measure of control over the level of detail actually produced.

With the System.Diagnostics namespace you can also write your own trace listener classes and add them to the Trace or Debug Listeners collection. When the CLR hits a Trace or Debug statement, all of the listeners in the collection will be called in turn. The DefaultTraceListener class, which provides debug tracing, is by default always in the collection. The Framework also comes with the EventLogTraceListener to dump these messages into the current log, and a TextWriterTraceListener to write them to streams.

Structured Exceptions

SEH is one of the best features of the .NET managed environment. From the perspective of Visual Basic 6.0, this is a fantastic improvement over On Error statements. You either catch an exception explicitly because you know what to do with it, or you don't bother and let someone else catch it higher up the chain.

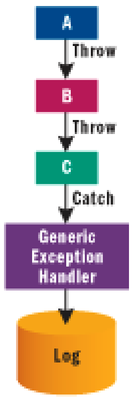

Figure 2 Log It

But if you use SEH just for this, you're missing out. You can implement your own base exception class from which all your other custom exceptions derive. In it you can capture data about the environment (such as the current user name or the thread ID) so that you have a complete log of everything that happened, allowing exceptional situations to be better evaluated and management reports to be generated as needed. **Figure 2** illustrates the process where an exception is propagated up the call chain to a generic handler that can then record it. You can log these any way you want, but a database usually makes the most sense (unfortunately, it's also the most work for you). If the exception logging mechanism is time intensive, it can also be done asynchronously to allow the application to continue execution while the exception information gets stored in the log.

WMI

WMI is by far the most powerful and sophisticated instrumentation mechanism included in Windows. Prior to the .NET Framework, it had a reputation for being difficult to use. This was largely due to the fact that the WMI API was practically impossible to use from RAD tools like Visual Basic 6.0. WMI is big—just take a look at the SDK documentation. Fortunately, you don't need to understand it all to make use of its best features. For example, add a reference to System.Management.dll. This will allow you to write the following code:

```
Imports System.Management.Instrumentation

<System.ComponentModel.RunInstaller(True), _
InstrumentationClass(InstrumentationType.Event)> _
Public Class MyEvent

    ' event "property"
    Public EventProperty As String

End Class
```

Even better is this code, which is more efficient than the declarative approach just shown:

```
<System.ComponentModel.RunInstaller(True)> _
Public Class MyEvent
             Inherits BaseEvent

    ' event "property"
    Public EventProp As String

End Class
```

You can raise the event just as easily:

```
Dim objEvent as MyEvent

objEvent = new MyEvent()
objEvent.EventProp = "Hello world!"
Instrumentation.Fire(objEvent)
```

This publishes an event to WMI which supports a loosely coupled publish/subscribe model. WMI allows you to raise events that will be delivered to their subscribers or, if none are running, that can be picked up later when one subscribes. This means that you don't have the overhead of actually publishing events if you know no one is listening.

One good reason to write your own listeners is to make use of logging facilities your company may already have, such as existing COM components that log to a central database which is used by system support personnel for all applications.

WMI is standards based; it's an implementation of the Distributed Management Task Force (DMTF), WBEM initiative, and the DMTF Common Information Model (CIM). Browse http://www.dmtf.org for more details. You'll see all of these acronyms as you dig around. WMI can interoperate with SNMP applications, making the enterprise vision of centralized monitoring an actual possibility. In the heterogeneous Web services world, a heterogeneous instrumentation capability is essential.

To start consuming these events, create a System.Management.ManagementEventWatcher with the appropriate WMI scope and query, and register an event handler for its EventArrived event. A simple event handler is shown in the code in **Figure 3**.

You can make this even easier using the WMI server explorer extensions (see "Managed WMI Extensions for Visual Studio .NET 2003 Server Explorer" in the Microsoft Download Center). You can just set up a query (in WQL, a subset of SQL) to your event, and drag and drop it onto a form. All the code I've shown is then generated for you.

The Good, the Bad, and the Ugly

Let's look now at when you should or shouldn't use each option by examining their strengths and limitations.

Event logging has a number of benefits. It's easy to use, it's durable, and you can view events from or log events to remote machines. You can even save off logs periodically for later investigation and reporting. Unfortunately, the disadvantages outweigh the benefits in most real-world situations. The event log is finite. You can set its size when the log is created, but then it's fixed. At that point, either your logging attempts throw an exception or you can configure the log to wrap, overwriting the earliest-written entries first. Because of this limitation, the event log is only useful for logging small amounts of information such as critical events like your application's start and stop messages or other high-level notifications. Additionally, it's also only available on Windows NT. If you still target Windows 98 or Windows Me, this facility isn't available.

Performance counters can be extremely useful. They provide excellent aggregated statistics on the health of your application. The number of clients connected, the number of database connections, and average page-turn times are ideal measurements for investigating load-related issues. Again, as with event logging, performance counters are not good for large amounts of specific details. Performance counters work using either average running values or absolute values.

Counters are helpful because Perfmon provides remote machine access and file logging. It even provides alerts, for example, running a program to send an e-mail message or Short Message Service (SMS) if your average response time exceeds an SLA.

Tracing is a real boon in the common language runtime (CLR) world. Built-in support provides out-of-the-box functionality and the .NET Framework allows you to use its supplied tracing handlers or to implement your own.

It should be noted that ASP.NET provides its own tracing facility separate from that provided by the System.Diagnostics namespace in the Framework. In ASP.NET, this is accessible through HttpContext.Current.Trace or through Page.Trace if you're code is running in a Web Form. The TraceContext class returned by these properties exposes two main methods for writing output messages: Write and Warn. All output is visible on the trace.axd page in your application's virtual directory.

The only difference between the two is that using the Warn method causes the message to appear on trace.axd in the log in red. You can also have ASP.NET dump the tracing output to the response stream after the current output has been rendered, causing the messages to be appended to the bottom of each page. This is only good for debugging, not for routine production instrumentation. The ASP.NET tracing system also logs additional useful information about each request. For example the current session ID, time, HTTP status code and headers, and cookies are shown.

By default, the System.Diagnostics tracing system and the ASP.NET tracing system are not linked in any way. Messages written to one do not get forwarded to the other. However, it is fairly simple to write a custom trace listener that forwards all trace and debugging messages to the ASP.NET trace. If you do this, just make sure to check that a current HttpContext exists.

There are a few other issues to be aware of when writing your own custom listeners. For example, if you make use of the performance counter or the event log and need to create a counter or an event source, you're going to need write access to the registry (as mentioned previously). This isn't practical or desirable for security reasons under the ASP.NET default account, so you have to be sure that your applications are installed and set up correctly prior to use. If possible, you should also configure your trace listeners using web.config rather than programmatically so that you can add and remove the listeners after deployment by changing your web.config or <*appname*>.config as follows:

```
<system.diagnostics>
    <trace autoflush="true" indentsize="0">
        <listeners>
            <add name="CustomListener",
                       type="MyLibrary.Class1,MyLibrary"/>
        </listeners>
    </trace>
</system.diagnostics>
```

The type attribute on the add element must be set to the listener's fully qualified type name. When using ASP.NET tracing, all trace calls are logged internally and can be displayed using a built-in IHttpHandler that exposes the tracing data of http://*yourservername*/*yourapp*/trace.axd.

The maximum number of requests logged here defaults to 10, but can be changed to another value in your web.config, like so:

```
<trace
        enabled="true"
        requestLimit="15"
        pageOutput="false"
        traceMode="SortByTime"
        localOnly="true"
/>
```

Unfortunately, with ASP.NET tracing you can't use your own tracing handler to process the output messages. The other, perhaps more significant, problem is that you can't assume you've got an HttpContext object available in the class libraries you develop. If you're developing generic class libraries that can be deployed on application server configurations, you always need to check whether HttpContext.Current is a valid reference before attempting to get the current TraceContext.

As soon as the requests exceed the RequestLimit attribute, you won't be logging your valuable tracing information anymore. This renders tracing for live problem diagnosis useless as there will likely be thousands of requests before the one you want to view, and then you still have no tools other than the provided page to track that request. The technique is really much more useful for debugging situations in which you want to check the correct flow control through your code.

A more general problem with ASP.NET tracing is that you need to explicitly turn it on in your configuration file. When you do this, however, you cause the application to be restarted. This is a bit heavy-handed just to configure your application's instrumentation. It is also a real issue if you're investigating a transient problem that actually requires the current application state.

Too much information can be a problem here too—when using diagnostics tracing, filtering isn't provided unless you code for it by using granular trace levels and trace switches. The amount of information you're likely to generate is overwhelming. This is especially true without a durable logging store, such as a file, and without a tool to process it usefully.

I've already talked about using SEH effectively. However, it is not meant to be an instrumentation tool. You'll end up bending your application code to throw exceptions for informational-type logging, which reduces the clarity that exceptions were designed to bring to CLR-based code. In addition, this hurts performance.

WMI is a mature instrumentation technology built into Windows 2000 and all subsequent versions. Since WMI is COM-based, you will be using interop, which can affect performance if used improperly. It also relies on DCOM for cross-machine logging and event correlation. WMI does, however, provide enterprise-scale facilities to perform many kinds of instrumentation. In fact, much of Windows itself is instrumented using WMI. For an effective instrumentation solution you need to be able to use the best of what's available and to make it configurable from your application. That way you can emit and consume the instrumentation you require according to the situation that presents itself. WMI brings you closer to this ultimate goal by unifying access to the instrumentation provided in your code.

A powerful feature of WMI is the ability to define an instrumentation contract for your application, called a schema. This schema defines the instrumentation you provide. It is specified in terms of event and data classes. .NET provides simple control over the schema content generated for your application. WMI is complementary to the logging schemes discussed already. For example, Event Log information and performance counters can actually be surfaced by WMI. This means that if you're already using these, you can make use of WMI now. The WMI SDK provides various tools, such as the browser-based CIM studio, so that you can examine your application's instrumentation (and query the schema). Many industrial-strength tools are based on WMI, such as Microsoft Operations Manager (MOM), HP OpenView, and IBM's Tivoli products. The WMI classes provided in the .NET System.Management namespace allow you to raise your own events and provide management facilities.

The System.Management namespace provides natural mappings onto WMI constructs. WMI classes become .NET Framework classes. Properties and attributes become fields and qualifiers. The data types in the .NET Framework map cleanly onto WMI data types, including strings and arrays.

Managed World

.NET now brings much greater power to all of these facilities with the EIF. This builds on WMI, making it easier to use. Although EIF dramatically reduces the overhead of WMI, it isn't included with the .NET Framework or Visual Studio®. Until now it hasn't been available unless you were an MSDN subscriber, so it hasn't exactly entered the mainstream. This is likely to change soon, especially with the new Logging Application Block released by the Microsoft Patterns and Practices group. [**Editor's Update - 3/26/2004:** EIF is now available for download at Microsoft Enterprise Instrumentation Framework.] WMI is the foundation for both of these and because it's been designed for scalability and doesn't require stopping your app for reconfiguration, it is a better choice than web.config/app.config-based solutions or #define TRACE and DEBUG.

EIF provides the Microsoft.EnterpriseInstrumentation namespace. There are two central constructs: event sources and sinks. An event source is the creator or publisher of an event. This is decoupled through configuration to the associated event sinks. Sinks consume and deal with events. This could be done, for example, by logging to the Windows Event Log or a database. Events in EIF are either implicit or explicit. Four implicit event sources are provided for you as classes: TraceMessageEvent, ErrorMessageEvent, AuditMessageEvent, and AdminMessageEvent.

To consume events, all event sinks must derive from the abstract EventSink class, which has a single static method, Raise. EIF provides three event sinks: Windows Trace, Event Log, and WMI. Windows tracing is implemented as a service, which runs in kernel mode to optimize performance. The Event Log sink directs events to the Windows Application Event Log, and the WMI sink uses WMI for logging events. The power of EIF is that the style of code you write to surface events is always the same. This makes it as easy as possible for you to start instrumenting your code. All you need to do is call the event's static Raise method, like so:

```
Imports Microsoft.EnterpriseInstrumentation
TraceMessageEvent.Raise("Hello World")
```

You can also write your own custom sinks to extend or add to the supplied ones. EIF is installer-aware and creates an EnterpriseInstrumentation.config file which you can edit manually or use the EIF API configuration classes.

Explicit events are those you define yourself, as follows:

```
<System.ComponentModel.RunInstaller(true)>

Public Class EIFEvent
Inherits Microsoft.EnterpriseInstrumentation.Schema.BaseEvent
    Public MyProperty as string  ' field
End Class
```

As you can see, it's almost identical to the WMI example shown earlier. This is because the class derives from the BaseEvent class which effectively replaces the BaseEvent you would use in WMI when developing your own event sources, as I've already discussed. The attribute defined ensures your event is installed in WMI when you run InstallUtil.exe against your .exe or.dll.

There are two configuration files needed to wire everything up. The EnterpriseInstrumentation.config is generated for you during installation and enables you to specify the relationships between sources and sinks and also filter events based on object type using categories. The TraceSessions.config file is needed to define tracing details only when you are using the Windows trace sink. I'll cover the use of TraceSessions.config in a minute, but let's first look at the salient features in the other one, shown in **Figure 4**.

The code in **Figure 4** has four sections. First, there is an <eventCategory> called All Events. As the <event> element specifies System.Object as the type, it includes all tag events raised by the System.Object type or any derived class—which means everything. The second element is the <filter>. This defines the event sinks that are wired up to the All Events category. An event raised will be dealt with by all three of the provided event sinks. The <eventSources> element is where you define the name of your instrumented application. This enables you to connect your application's events to one or more sinks through the <filterBindings> element. It defines the relationship between your sources and sinks. This example uses the implicit Application event source that EIF provides for your whole application. It is the same source name used in the earlier code samples.

The default TraceSessions.config file resides in the folder [EI_Dir]\Bin\Trace Service, where [EI_Dir] is the location in which you installed EIF. You can use the default session name, TraceSession, or you can add your own session name. See the EIF documentation for further details.

WMI and EIF applications can be a bit cumbersome to configure and get working properly. You may also find you get a few fatal exceptions during development due to the managed/unmanaged interaction that takes place.

Logging Application Block builds on WMI and EIF by providing common facilities that developers need, such as database and queue-based logging. See the Patterns and Practices site for more details.

A Strategy

WMI and its dependent technologies offer abstraction by decoupling the incidence of instrumentation from the action of handling it. EIF goes much further than WMI, though. Not only are the source and sinks decoupled, but standard sinks are provided for you for common requirements. Without EIF, you are deciding what types of instrumentation you want to use and coding specifically for them. If you want event logging, for example, you use System.Diagnostics.EventLog as detailed earlier. But what if your application grows in size and you want to move your instrumentation logging to a database? You've got a big find-and-replace operation on your hands. With EIF, you can just change the EnterpriseInstrumentation.config to use a different event sink and you're done—if you use Logging Application Block you don't even need to write this sink yourself since there's one provided.

You need to think about how best to make use of each of the available facilities. Each of the available features in Windows is very specific and you need them all in order to be effective. This might seem like a lot of work even with the .NET Framework, but it's not. Judicious use of each will yield the best results and require the least amount of work. **Figure 5** details a few guidelines regarding when to use each one.

The first thing here is order. Sprinkling code for exceptions and tracing is never effective. You must think about what instrumentation to add as you are coding. Error and exception handling are exactly the same in that you can't retrofit this stuff effectively. When you write code, you know exactly what your intentions are—the semantics. Everything's fresh in your mind, the task is at hand. This is the best time to add your exceptions, error handling, and instrumentation because you are thinking about what can go wrong with each line of code. You are also thinking about what would be useful to measure. Doing this at the design and modeling phase would be even better, but often this level of detail is hard to attain before starting to code so you should always be pragmatic.

Once you have added instrumentation code, you find that it actually gives you more than just simple text logging to various places. Its flexibility gives you an extensibility point. The implementation of your trace code is able to change as you add new features without affecting any of the actual application code. Just as with writing a good base, SEH is critical before you start coding. Doing the same with instrumentation will yield benefits that will be reusable for future projects as well.

For example, you might want to determine method timings. If in each function you have a trace in/out and exception path, you can measure the time between them. Note this is a different scheme from aspect-oriented programming (AOP), which is limited to intercepting calls and has no knowledge of what those calls do. Here you are explicitly adding that information to help you later. Or you might want to filter the instrumentation. You might only want to record the fact that you are receiving SOAP messages greater than 100KB in size. Although not an error, this is something you want to track. So you can add a specific log point with this information, which your generic handler can track for you. You can achieve this easily by implementing your own custom event sink.

Security

Logging information brings its own security risks. If you start instrumenting sensitive data, such as database connection strings, you risk sending it in the clear across your network. With distributed applications this can mean a big security hole.

As mentioned earlier, a certain level of registry access is required in order to use performance counters, Windows Event Logging, and WMI from your code. A more general problem, though, is accessing the instrumentation data once it's been recorded. If your production applications run on an isolated network with demilitarized zones (DMZs), firewalls, and the like, chances are you won't be able to remove the logs from your corporate network. This makes it even more important that you log all your information to a database so that you can be granted read privileges when necessary for problem investigation.

Conclusion

When asked why an application isn't better instrumented, many designers and developers cite performance as the key concern. Never use performance as an excuse for not providing adequate instrumentation. This argument just doesn't hold up in most cases when you account for the processing time of business logic. Good instrumentation can save many hours of late-night frustration and keep the wheels of your enterprise applications running smoothly. I've only covered the fundamentals of instrumentation here. I'm sure you can think of ways to expand on the ideas I've presented. Gaining an understanding of the new features of the .NET Framework is well worth the effort. Spend the time to take a closer look and see how you can apply what I've shown in your next project.

Jon Fancey runs an IT consultancy company in the United Kingdom that specializes in enterprise-class software design and implementation. You can reach him at instrumentation@jonfancey.com.

From the April 2004 issue of MSDN Magazine.
Get it at your local newsstand, or better yet, *subscribe*.

Figure 3 Event Handler

```
Imports System.Management

Private Sub MyEventArrivedHandler(
        ByVal sender As Object, ByVal evt As EventArrivedEventArgs)

    Console.WriteLine("Event received with the following properties:")

    ' Loop through all of the ManagementBaseObject's properties
    For Each eventProperty In evt.NewEvent.Properties
        Console.WriteLine("Property {0}: {1}",
            eventProperty.Name, eventProperty.Value)
    Next eventProperty

EndSub
```

Figure 4 Config File

```
<eventCategories>
    <eventCategory name="All Events" description="A category that contains
    all events.">
        <event type="System.Object" />
    </eventCategory>
</eventCategories>

<filters>
    <filter name="defaultSoftwareElementFilter" description=" A default filter
    for the Software Element event sources.">
        <eventCategoryRef name="All Events">
            <eventSinkRef name="wmiSink" />
            <eventSinkRef name="traceSink" />
            <eventSinkRef name="logSink" />
        </eventCategoryRef>
    </filter>
</filters>

<eventSources>
    <eventSource name="Application" type="softwareElement"
    internalExceptionHandler="ignore" description="Application level event
    source." />
</eventSources>

<filterBindings>
    <eventSourceRef name="Application">
        <filterRef name="defaultSoftwareElementFilter" />
    </eventSourceRef>
</filterBindings>
```

Figure 5 Instrumentation Options

Feature	**Positive Attributes**	**Negative Attributes**
Event log	High-level application events (for example start and stop)	Limited in number of events
Performance counters	Use when you have rollup summary information (number of active users) or absolute (simple) values you want to measure (total requests)	Not suitable for complex data
Debug output	Single user code pathing and debugging	Overwhelming if used everywhere
SEH	Good for unexpected exceptions	Can bend your design
System.Diagnostics.Trace	Easy to use	Limited to Web apps
ASP.NET trace	Great for debugging	Fixed functionality
WMI	Widely supported, complementary to other schemes (for example, your event log code can be surfaced through WMI)	More complex to use and install applications with
EIF	Easier to program than WMI	Doesn't expose all of WMI's power
LAB	Common requirements provided: MSMQ logging, DBMS logging	A lot to learn