Microsoft

Exception Management in .NET

patterns & practices

Version 1.0

Contents

Exception Management in .NET

Exception Management in .NET

Exception Management

To build successful and flexible applications that can be maintained and supported easily, you must adopt an appropriate strategy for exception management. You must design your system to ensure that it is capable of the following:

- Detecting exceptions.
- Logging and reporting information.
- Generating events that can be monitored externally to assist system operation.

Spending the time at the beginning to design a clear and consistent exception management system frees you from having to piece one together during development, or worse still, from having to retrofit exception handling back into an existing code base.

An exception management system should be well encapsulated and should abstract the details of logging and reporting from the application's business logic. It should also be capable of generating metrics that can be monitored by operators to provide an insight into the current health and status of the application. This helps to create an application that can quickly and accurately notify operators of any problems it is experiencing, and can provide valuable information to assist developers and support services with problem resolution.

Understanding Exceptions and Exception Hierarchies

The Microsoft® .NET common language runtime provides a means for notifying programs of exceptions in a uniform way, even if the module generating the exception is written in a different language than the one handling the exception.

Exceptions represent a breach of an implicit assumption made within code. For example, if your code tries to access a file that is assumed to exist, but the file is missing, an exception would be thrown. However, if your code does not assume that the file exists and checks for its presence first, this scenario would not necessarily generate an exception.

Exceptions are not necessarily errors. Whether or not an exception represents an error is determined by the application in which the exception occurred. An exception that is thrown when a file is not found may be considered an error in one scenario, but may not represent an error in another.

All exception classes should derive from the **Exception** base class within the **System** namespace for compliance with Common Language Specification (CLS). The .NET Framework class library) provides an extensive hierarchy of exception classes to handle various types of common exceptions, all of which ultimately derive from **Exception**. Exceptions can be generated by the .NET runtime or they can be created programmatically. You will find more information on how to create your own exception hierarchies later in this document.

ApplicationException serves as the base class for all application-specific exception classes. It derives from Exception but does not provide any extended functionality. You should derive your custom application exceptions from ApplicationException. Figure 1 illustrates the basic exception class hierarchy.

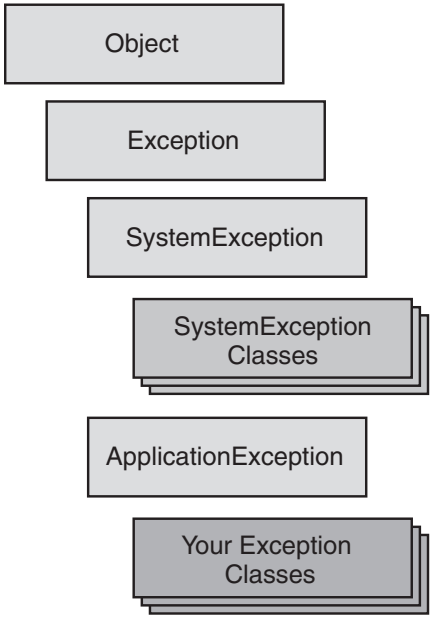

Figure 1
Exception class hierarchy

Exception Handling Process

There are two main processes for handling exceptions. The process flowchart shown in Figure 2 illustrates the basic steps that your application should perform to handle an exception. Each method or procedure within your application code should follow this process to ensure that exceptions are handled within the appropriate context defined by the scope of the current method.

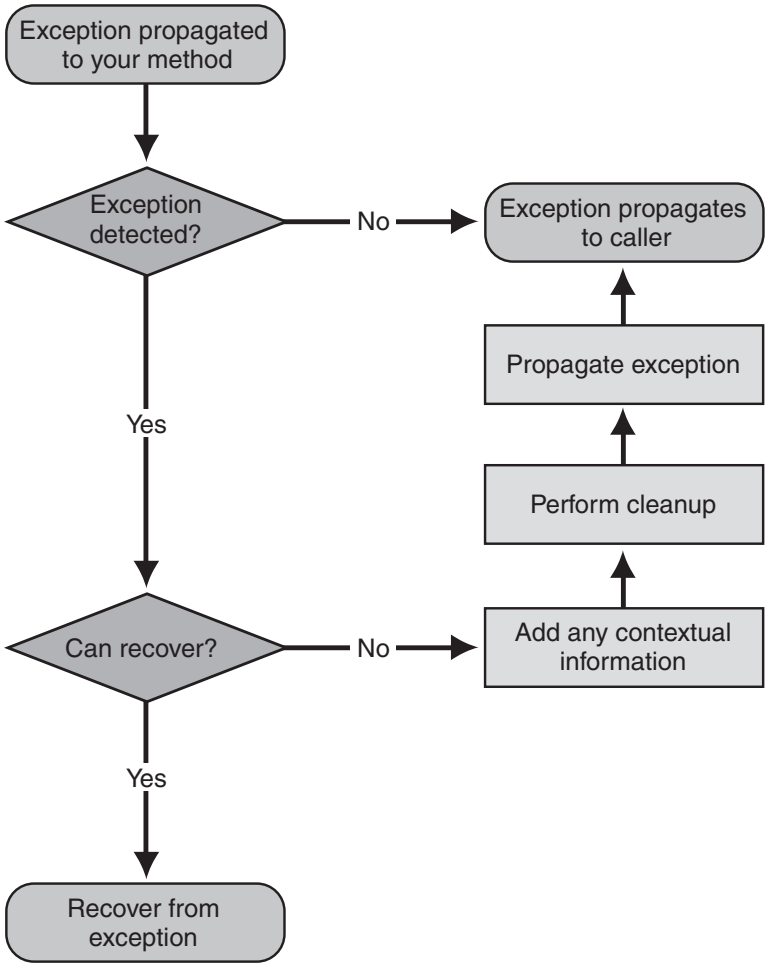

Figure 2
Exception handling process

This process continues to occur as an exception propagates up the call stack. The "Exception Detection" and "Exception Propagation" sections discuss this process in detail. Figure 3 shows the process your application should perform when the exception is propagated to the last point or boundary at which your application can handle the exception before returning to the user.

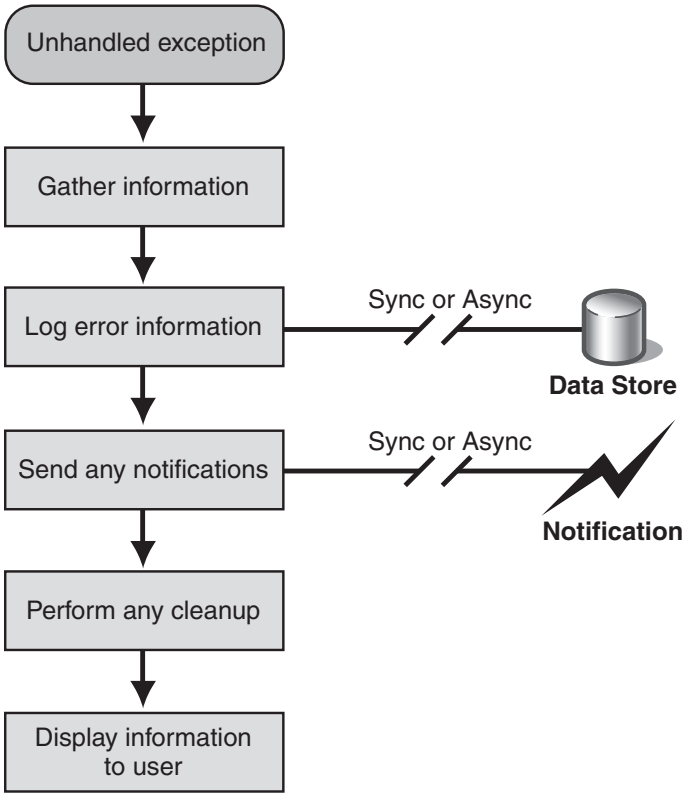

Figure 3
Processing of unhandled exceptions

This process is vitally important to your application's ability to persist exception information, notify operational resources of problems, and properly manage the user experience. This process is discussed in the Managing Unhandled Exceptions, Gathering Information, Logging, and Notification sections of this document.

Exception Detection

The .NET Framework allows all .NET languages to take advantage of structured exception handling. Structured exception handling provides a control structure that includes exceptions, protected blocks of code, and filters to allow your code to handle exceptions robustly and efficiently. You can use **try**, **catch**, and **finally** blocks to detect exceptions thrown within your code and to react to them appropriately.

```
try
{
   // Some code that could throw an exception.
}
catch(SomeException exc)
{
   // Code to react to the occurrence
   // of the exception
}
finally
{
   // Code that gets run always, whether or not
   // an exception was thrown. This is usually
   // clean up code that should be executed
   // regardless of whether an exception has
   // been thrown.
}
```

The **try** block contains code that could throw an exception. When an exception is thrown within this block, the first **catch** block whose filter matches the class of the exception catches it. Note that the filter is specified within the parentheses following the **catch** keyword. If you have multiple **catch** blocks, you should ensure that these are ordered from the most specific type to the most generic type. This ensures the most specific type **catch** block is executed for any given exception. The **finally** statement executes, allowing clean up and other code to be executed, regardless of whether an exception is thrown.

You should only catch exceptions when you need to specifically perform any of the following actions:

- Gather information for logging
- Add any relevant information to the exception
- Execute cleanup code
- Attempt to recover

If a particular method does not need to perform any of these actions, it should not catch the exception; rather, it should allow it to propagate back up the call stack. This keeps your code clean and explicit as you only catch the exceptions that need to be handled within the scope of a particular method and allow all others to continue to propagate.

Use Exceptions Appropriately

You should only throw exceptions when a condition outside of your code's assumptions occurs. In other words, you should not use exceptions as a means to provide your intended functionality. For example, a user might enter an invalid user name or password while logging on to an application. While this is not a successful logon, it should be a valid and expected result, and therefore should not throw an exception. However, an exception should be generated if an unexpected condition occurs, such as an unavailable user database. Throwing exceptions is more expensive than simply returning a result to a caller. Therefore they should not be used to control the normal flow of execution through your code. In addition, excessive use of exceptions can create unreadable and unmanageable code.

Exceptions Intercepted by the Runtime

In certain scenarios, an exception you throw may be caught by the runtime and an exception of another type might be thrown up the call stack in place of the original. For example, consider the case in which you call the **Sort** method on an **ArrayList** of your objects. If one of your objects throws an exception in the **CompareTo** method of its **IComparable** interface, the exception is caught by the runtime and a **System.InvalidOperationException** exception is thrown to the code that calls the **Sort** method. In addition, any exception thrown by a method that you invoke through reflection will be caught by the runtime and a **System.Reflection.TargetInvocationException** will be thrown to your code. In these scenarios your original exception is not lost. It is set as the **InnerException** of the exception thrown by the runtime (the **InnerException** property is discussed in the "Exception Propagation Section"). Situations like this can occur in a number of scenarios. You should be aware of this and test your applications thoroughly to minimize the impact of these scenarios.

For more information on these topics, see the following:

- Base Exception Hierarchy
- Common Exception Classes
- System.Exception Class

Exception Propagation

There are three main ways to propagate exceptions:

- **Let the exception propagate automatically**

  With this approach, you do nothing and deliberately ignore the exception. This causes the control to move immediately from the current code block up the call stack until a **catch** block with a filter that matches the exception type is found.

- **Catch and rethrow the exception**

  With this approach, you catch and react to the exception, and clean up or perform any other required processing within the scope of the current method. If you cannot recover from the exception, you rethrow the same exception to your caller.

- **Catch, wrap, and throw the wrapped exception**

  As an exception propagates up the call stack, the exception type becomes less relevant. When an exception is wrapped, a more relevant exception can be returned to the caller. This is illustrated in Figure 4. With this approach, you catch the exception, which allows you to react to it, clean up, or perform any other required processing within the scope of the current method. If you cannot recover, wrap the exception in a new exception, and throw the new exception back to the caller. The **InnerException** property of the **Exception** class explicitly allows you to preserve a previously caught exception. This allows the original exception to be wrapped as an inner exception inside a new and more relevant outer exception. The **InnerException** property is set in the constructor of an exception class.

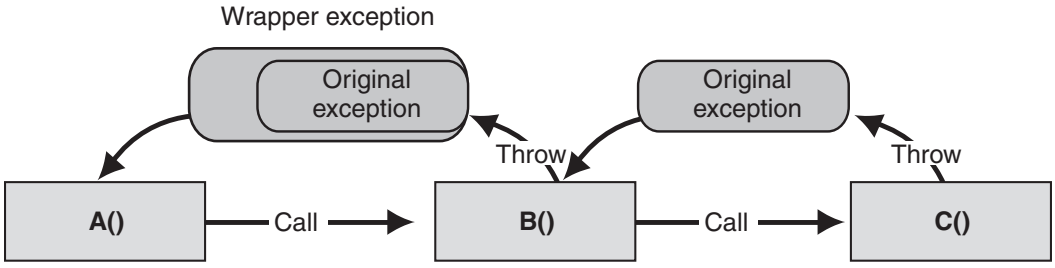

Figure 4
Exception propagation

As the exception propagates up the call stack, **catch** blocks can only catch the outer exception. Inner exceptions are programmatically accessible through the **InnerException** property, but they cannot be matched to a **catch** block.

The following example shows how these methods are implemented.

```
try
{
   // Some code that could throw an exception.
}

catch(TypeAException e)
{
   // Code to do any processing needed.
   // Rethrow the exception
   throw;
}
```

```
catch(TypeBException e)
{
   // Code to do any processing needed.

   // Wrap the current exception in a more relevant
   // outer exception and rethrow the new exception.
   throw(new TypeCException(strMessage, e));
}

finally
{
   // Code that gets executed regardless of whether
   // an exception was thrown.
}
```

The first **catch** block catches an exception of type **TypeAException**, performs any processing needed, and rethrows the same exception back up the call stack. The second catch block shows how you could wrap the **TypeBException** in a more relevant outer exception of type **TypeCException** and throw the new outer exception up the call stack. In this example the code only cares about exceptions of either the **TypeAException** or **TypeBException** type. For all other exception types, the code lets the exception propagate automatically.

Table 1 lists the three approach to propagate exceptions and the benefits and limitations of each approach.

Table 1. Exception Propagation Summary

Means of Propagation	Allows you to react to the exception	Allows you to add relevancy
Let the exception propagate automatically	No	No
Catch and rethrow the exception	Yes	No
Catch, wrap, and throw the wrapped exception	Yes	Yes

When to Use InnerException

You should wrap an exception only when there is a compelling reason to do so. When an exception is initially thrown, it provides information about the exact cause of the exception. As an exception propagates up the call stack, the exception type becomes less relevant. Wrapping an exception can provide a more relevant exception to the caller.

For example, consider a hypothetical method called **LoadUserInfo**. This method may load a user's information from a file that is assumed to exist when the method

tries to access it. If the file does not exist, a **FileNotFoundException** is thrown, which has meaning within the context of the **LoadUserInfo** method. However, as the exception propagates back up the call stack—for example, to a **LogonUser** method—a **FileNotFoundException** exception does not provide any valuable information. If you wrap the **FileNotFoundException** in a custom exception class (discussed later in this document)—for example, one called **FailedToLoadUserInfoException**—and throw the wrapper exception, it provides more information and is more relevant to the calling **LogonUser** method. You can then catch the **FailedToLoadUserInfoException** exception in the **LogonUser** method and react to that particular exception type rather than having to catch the **FileNotFoundException**, which is an implementation detail of another method.

Custom Exceptions

The .NET Framework is a type-based system that relies on the exception type for identification, rather than using method return codes such as HRESULTs. You should establish a custom hierarchy of application-specific exception classes that inherit from **ApplicationException**, as illustrated in Figure 5.

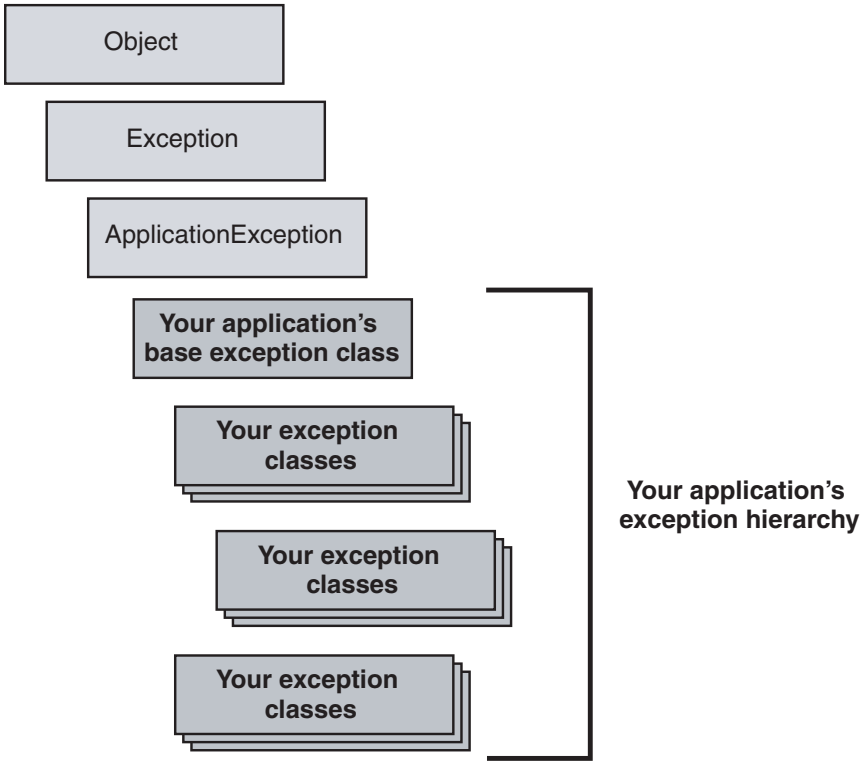

Figure 5
Application exception hierarchy

This hierarchy allows your application to benefit from the following:

- Easier development because you can define properties and methods on your base application exception class that can be inherited by your other application exception classes.
- New exception classes created after your application has been deployed can derive from an existing exception type within the hierarchy. Existing exception handling code that catches one of the base classes of the new exception object will catch the new exception without any modifications to the existing code, interpreting it as one of the object's base classes.

Designing Your Application Exception Hierarchy

The .NET Framework provides an extensive hierarchy of exception classes. If a suitable exception class is already provided by the .NET Framework, use it instead of creating a new class. You should only create a new application exception class for an exception type that you need to react to and handle within your code that is not already available in your application exception hierarchy or in the Framework. Most application exception hierarchies should be fairly flat with grouping used for organization purposes or to allow some set of application exceptions to inherit common properties or functionality.

As you create your hierarchy, use the following questions to help you to decide if you need to create a new exception class.

- **Does an exception exist for this condition?**
  If an exception exists either within your current hierarchy or within the Framework, use it instead of creating your own exception class.
- **Does a particular exception need discrete handling?**
  If so, you should create an exception class to allow your code to catch the specific exception and handle it explicitly. This eliminates the need to catch a more generic exception and then use conditional logic to determine what action to take.
- **Do you need specific behavior or additional information for a particular exception?**
  If so, you can use a new application exception class to include additional information or functionality to suit a specific need.

You should store your application exception hierarchy in a single assembly (or small group of assemblies for larger systems) that can be referenced throughout your application code. This helps to centralize the management and deployment of your exception classes.

Creating a Custom Exception Class

To ensure standardized naming, you should always end your custom exception class names with the word **Exception**. It is also good practice for each of your custom exception classes to provide the three constructors shown in the following class definition:

```
using System;
public class YourBaseApplicationException : ApplicationException
{
   // Default constructor
   public YourBaseApplicationException ()
   {
   }

   // Constructor accepting a single string message
   public YourBaseApplicationException (string message) : base(message)
   {
   }

   // Constructor accepting a string message and an
   // inner exception which will be wrapped by this
   // custom exception class
   public YourBaseApplicationException(string message,
            Exception inner) : base(message, inner)
   {
   }
}
```

Creating a Base Application Exception Class

Your application should have a single base exception class that derives from **ApplicationException**. This class serves as the base class for all of your application's exception classes, as shown earlier in Figure 5.

You should add fields to this base class to capture specific information, such as the date and time the exception occurred, the machine name, and so on. This encapsulates the common exception details into the single base class and makes this information available to your exception classes through inheritance.

Remoting Custom Exceptions

The **Exception** class implements the **ISerializable** interface, allowing it to manage its own serialization. To allow your exceptions to be marshaled across remoting boundaries, you need to attribute your class with the **`[Serializable]`** attribute and include the additional constructor shown below.

```
protected YourBaseApplicationException(SerializationInfo info,
      StreamingContext context) : base(info, context)
{
}
```

If your exceptions add fields to the base **ApplicationException** class, you will need to persist these values programmatically into the serialized data stream. This can be done by overriding the **GetObjectData** method and adding these values to the **SerializationInfo** object as shown below.

```
[Serializable]
public class ExampleException : ApplicationException
{
   public ExampleException() : base()
   {
   }
   public ExampleException(string message) : base(message)
   {
   }
   public ExampleException(string message,Exception inner) :
                  base(message,inner)
   {
   }
   protected ExampleException(SerializationInfo info, StreamingContext context) :
                  base(info,context)
   {
      m_strMachineName = info.GetString("m_strMachineName");
   }

   public override void GetObjectData( SerializationInfo info,
                  StreamingContext context )
   {
      info.AddValue("m_strMachineName", m_strMachineName,
                  typeof(String));

      base.GetObjectData(info,context);
      }

   private string m_strMachineName = Environment.MachineName;

   public string MachineName
   {
      get
      {
         return m_strMachineName;
      }
      set
      {
         m_strMachineName = value;
      }
   }
}
```

The values can be retrieved from the **SerializationInfo** object in the constructor of your exception object using **info.GetValue** or one of the **Get** methods of the **SerializationInfo** object (such as the **GetString** method shown above). Your custom exceptions should be able to maintain their state as they are marshaled across local and remote boundaries. Even if your application does not currently span remoting boundaries, providing serialization support ensures that exception information will not be lost if your application is later modified to do so. If you do not programmatically serialize your exception's custom values, they will not be set properly when they are deserialized at the client. Providing serialization support also allows your application to serialize all the information about your exception and store it for logging purposes.

For more information on these topics, see the following:

- .NET Framework Developer's Guide: Best Practices for Handling Exceptions
- Handling and Throwing Exceptions
- Implementing ISerializable
- Serialization Sample

Managing Unhandled Exceptions

When an unhandled exception propagates to the last point or boundary at which your application can handle the exception before returning to the user, your application can no longer recover from the exception. At this point, it must gather information, log information, send any notifications, perform any cleanup, and do any other processing needed before managing the communication of the exception condition to the user. In most Web-based applications, this boundary is controlled by your ASP.NET code through either Web pages to end users or Web services to other applications. This section discusses the tools available to manage unhandled exceptions at the system boundary.

ASP.NET

With ASP.NET, you are no longer restricted by the limited capabilities of scripting languages. ASP.NET code is compiled and can be written in any .NET Framework language. As opposed to the limited techniques available in the traditional ASP world, ASP.NET allows your pages to use all of the exception handling techniques that your components can. ASP.NET provides some specific functionality to allow your application to manage exceptions and configure how information should be displayed back to the user.

Configuring Web.config Settings

You should configure exception management settings within your application's Web.config file. The following is an example of the exception settings in a Web.config file.

```
<customErrors defaultredirect="http://hostname/error.aspx" mode="on">
  <error statuscode="500" redirect="/errorpages/servererror.aspx" />
  <error statuscode="404" redirect="/errorpages/filenotfound.htm" />
</customErrors>
```

In the **customErrors** element, specify a default redirect page. There are three modes for the default redirect page:

- **on**

  Unhandled exceptions will redirect the user to the specified defaultredirect page. This is used mainly in production.

- **off**

  Users will see the exception information and not be redirected to the defaultredirect page. This is used mainly in development.

- **remoteonly**

  Only users accessing the site on the local machine (using localhost) will see the exception information while all other users will be redirected to the defaultredirect page. This is used mainly for debugging.

In addition to the default redirect page, you can set specific pages for certain HTTP error codes. For example, you can specify that all 404 errors result in a certain error page, while all 500 errors result in another (as shown above).

Before .NET was available, these settings had to be configured within the Internet Information Services (IIS) metabase. The ASP.NET Web.config file centralizes the configuration settings for your application into a file, allowing xcopy deployment of your application together with its settings.

You should be aware that these settings only apply to ASP.NET files (that is, files with specific extensions, such as .aspx and .asmx). For example, if a user requests an .htm or .asp file that does not exist on your Web server, IIS will redirect the user based on the HTTP error settings defined within the IIS metabase and not the Web.config file. If you want consistency between ASP.NET and non-ASP.NET files, you should configure IIS and the Web.config file to redirect users to the same pages. Any settings in the IIS metabase must be deployed along with your application files.

Using the @ Page Directive

The Web.config file settings apply to its directory and any child directories. These settings can be overridden for specific pages using the **ErrorPage** attribute of the **Page** directive. For example:

```
<%@ Page ErrorPage="customerror.aspx" %>
```

This sample will redirect the user to **customerror.aspx** if an unhandled exception occurs within the current page regardless of the Web.config settings.

Handling ASP.NET Exceptions

ASP.NET provides two main events for reacting to unhandled exceptions that propagate up from your application code:

- **Page_Error**

  This event is fired when an exception is not handled at the page level. In order for this event to be fired, you should include a **Page_Error** event handler in your page's code and either verify that the AutoEventWireup attribute of the @ Page directive is set to true (the default), or manually hook up the event using the following code.

  ```
  Page.Error += new System.EventHandler(Page_Error);
  ```

- **Application_Error**

  This is located in the global.asax file and as a result has application-wide scope. It occurs if an exception is not handled by a particular page. You should use this event to do any logging, notifications, and any other processing needed before you gracefully return an error to the user.

  ```
  // In global.asax file
  protected void Application_Error(Object sender, EventArgs e)
  {
        Exception exc = Server.GetLastError();
        // Perform logging, send any notifications, etc.
  }
  ```

In these events, you use the **Server.GetLastError** method to obtain a reference to the unhandled exception object. A **Server.ClearError** can also be used to clear the exception and stop any further propagation. For example, your page might have a **Page_Error** event that uses **Server.GetLastError** to access the unhandled exception object. If you do not clear the exception by using the **Server.ClearError**, the unhandled exception will continue to propagate up to the **Application_Error** event. If you use **Server.ClearError** to clear the exception in **Application_Error**, the **defaultredirect** setting in the Web.config file will not redirect the user because the unhandled exception no longer exists. If you want the **defaultredirect** setting to properly redirect users, do not clear the exception in **Application_Error**.

For more information on these topics, see the following:

- ASP.NET Homepage
- <customerrors> Section

Web Services

Web services are limited in their ability to propagate exceptions using the Simple Object Access Protocol (SOAP). The .NET Framework provides the **SoapException** class as the only means to raise exceptions using SOAP. The CLR throws a **SoapException** automatically when it receives a malformed SOAP request from the client.

When a Web service method throws any unhandled exception, the exception is repackaged as a **SoapException** and is returned to the Web service client via the SOAP response. The **ServerFaultCode** is used to indicate that an error that was not a result of the message format or contents occurred during the processing of the request. The SOAP response that the server runtime creates is in a standard SOAP format allowing non-.NET clients to detect that an error has occurred. On a .NET-based client, the Framework captures the SOAP message and deserializes the **SoapException** so that the client can detect exceptions through standard exception handling techniques, just as it would for a non-Web service call. However, the exception that is detected on the client is a **SoapException**, not the exception type that was originally thrown by the Web service. Information about the original exception is included in the Message property of the **SoapException**.

Hiding Exception Information

For Web services that are exposed to external companies and systems, you may not want to expose all of your exception information to your clients. You should throw an exception to your clients so that they can react, but you may not want all of the details of your specific exception to be sent outside of your company. You may only want to throw an exception that indicates that the service is currently experiencing problems.

In this case, you should create a generic application exception that contains any information you want to convey. After catching an unhandled exception in your Web service, you should log the exception and perform any necessary processing, then construct a new instance of the generic application exception. You can then set any desired information in the generic application exception and throw it to the client. This allows you to log detailed information in the Web service and throw a less detailed exception to your clients.

For more information, see the **SoapException** Class documentation in the .NET Framework Class Library.

The UnhandledExceptionEventHandler Delegate

The **UnhandledExceptionEventHandler** delegate can be used as a means to handle exceptions that are not handled by your application code. The method you associate with the **UnhandledExceptionEventHandler** delegate will be executed when an unhandled exception occurs.

In most ASP.NET Web applications, your application will use **Application_Error**, instead of the **UnhandledExceptionEventHandler**, to serve as your application's last chance to handle the exception.

For more information, see the **UnhandledExceptionEventHandler** Delegate documentation in the .NET Framework Class Library.

Gathering Information

You should ensure that you always capture all the appropriate information that accurately represents the exception condition. What you do with this information depends on the needs of the recipient of the information. Potential recipients of this information include the following:

- **End users**

  They require a meaningful and well presented description.
- **Application developers**

  They require more detailed information to assist with problem diagnosis.
- **Operators**

  They require relevant information that allows them to react appropriately and take the necessary recovery steps.

Capturing Appropriate Information

It is important that each group receive the information that is relevant to them, allowing them to react appropriately. Not all information is appropriate for all audiences. For example, your users should not see cryptic messages that include source code line numbers or conditions relating to programmatic conditions, such as an index being out of range. Equally annoying for end users are generic messages such as "Error Occurred. Please contact the system administrator." Your users should be shown a helpful message that includes any actions that they can take to rectify the problem. Always capture all of the relevant information and filter the display of this information based on the recipient's needs. Table 2 on the next page provides a summary of required information by audience level.

Table 2. Exception Message Information by Audience

Audience	Required Information
End users	An indication of whether the request succeeded. A well presented message indicating what went wrong. Instructions telling them what they should do to rectify the problem.
Application developers	Date and time that the exception occurred. The precise location of the source of the exception. Exactly which exception occurred. Information associated with the exception and the state of the system when the exception occurred.
Operators	Date and time the exception occurred. The precise location of the source of the exception. Exactly which exception occurred. What resources should be notified and what information needs to be given. The exception type indicating whether it relates to an operational or development issue.

To allow your application to provide rich information that can be tailored to suit the various groups, you should capture the details presented in the Table 3. Note that the Source column indicates the class and the class member that can generate this information. Unless otherwise noted, all classes are located within the **System** namespace.

Table 3. Exception Information to Capture

Data	Source
Date and time of exception	DateTime.Now
Machine name	Environment.MachineName
Exception source	Exception.Source
Exception type	Type.FullName obtained from Object.GetType
Exception message	Exception.Message
Exception stack trace	Exception.StackTrace—this trace starts at the point the exception is thrown and is populated as it propagates up the call stack.
Call stack	Environment.StackTrace—the complete call stack.
Application domain name	AppDomain.FriendlyName
Assembly name	AssemblyName.FullName, in the System.Reflection namespace
Assembly version	Included in the AssemblyName.FullName
Thread ID	AppDomain.GetCurrentThreadId
Thread user	Thread.CurrentPrincipal in the System.Threading namespace

Accessing All Exception Data

Exceptions can have various structures and expose different public properties that offer valuable information about the cause of the exception. It is important to capture all of this information in order to obtain a complete picture of what went wrong. By using reflection (implemented in the **System.Reflection** namespace) you can easily walk the public properties of an exception and access all of its values. This is useful for generic logging routines that accept an argument of type **Exception**. Such routines can use reflection to interrogate and access the detailed and specific exception information. Sufficient exception detail is critical to allow developers to determine the source of the exception.

For more information on these topics, see the following:

- Reflecting on Stack Traces
- System.Reflection Namespace

Application Instrumentation

Instrumentation is the act of incorporating code into your program that reveals application-specific data to someone monitoring that application. Raising events that help you to understand an application's performance or allow you to audit the application are two common examples of instrumentation.

The Enterprise Instrumentation framework (EIF) provides a flexible way of handling many application events, including exceptions. It uses event sources and event sinks to decouple an application's logging functionality from its notification delivery system. However, for exception handling specifically, it is also possible to build custom logging and notification solutions using the various techniques discussed in the "Logging" and "Notification" sections of this document. Each of these options for instrumentation are discussed in turn.

Enterprise Instrumentation Framework (EIF)

EIF is a comprehensive instrumentation application programming interface (API) especially targeted at multi-tier .NET applications that may scale to multiple application servers. EIF provides a unified programming model for publishing application events. It is recommended for instrumenting enterprise applications.

EIF benefits include the following:

- EIF greatly simplifies the coding required to publish application events.
- EIF simplifies coordination between system developers, application developers, and operations staff by using an *event schema* that defines what events may be raised by the application and what data they will contain.

- EIF is a single, uniform interface that builds upon Windows Management Instrumentation (WMI), the Windows Event Log service, and Windows Event Trace. This allows for easier migration between the various reporting mechanisms. Also, application developers need not become familiar with the details of each of these interfaces.
- EIF handles many different types of events, such as security-related events and error-related events.
- The operations team can configure EIF applications. They can select which events they are interested in and filter out others. This frees developers from modifying code to support the changing monitoring needs of the operations staff.

To use EIF, you must have the following components installed on your development workstation:

- You must be running Windows XP Professional or Windows 2000 with service pack (SP) 2 or later installed (all editions. However, if you are using Windows 2000, it is recommended that you upgrade to SP3 or later.
- The .NET Framework needs to be installed so that EIF can install and run. It is compatible with both .NET Framework version 1.0 SP2 or later and.NET Framework 1.1.
- Microsoft Visual Studio® .NET Enterprise Architect, Visual Studio .NET Enterprise Developer, or Visual Studio .NET Professional (or later) are recommended as an authoring environment but are not required for instrumented applications to run.

You can obtain EIF through Enterprise and Universal MSDN® Subscriptions.

EIF Basics

EIF introduces *event sources* and *event sinks* that decouple events being raised from the underlying eventing, tracing, or logging mechanism. In Enterprise Instrumentation, events are always raised from a specific event source. The event source configuration ultimately determines whether the event is raised or not, what information the event contains, and to which eventing, tracing, or logging mechanisms the event is routed. In all cases, information about the event source, such as event source ID, is passed as property values in the event object.

An event sink is responsible for taking an event and firing or logging it using a specific mechanism, such as WMI, the Windows Event Log service, or Windows Event Trace

The EIF configuration file, which is written in XML and is external to the application code, allows you to map connections between event sources and event sinks. This feature provides a great deal of flexibility. It allows operations staff to specify what events will be raised, what data fields will be included, and which event sink(s) will receive the data.

The format of the data passed between event sources and event sinks is given by an *event schema*. The event schema lists all of the event types that may arise from a given application.

EIF Event Schema

The standard event schema is a starting point for defining the set of events that an application (or all applications) can publish. The following are the types of events defined in that schema (but you will often want to add to these or create entirely new kinds of events):

- Trace events monitor the progress of program execution. They are placed at the programmer's discretion.
- Notification events log events, such as the use of secure resources or application-level warnings.
- Error events signal abnormal conditions or conditions that require user intervention.

Using EIF to Add Instrumentation

This section gives a conceptual overview of how to add instrumentation to your application using EIF. More detailed information can be found in Chapter 4 of the *.NET Operations Guide* and in the EIF documentation.

There are two steps to adding EIF instrumentation to your application. The first is to define one or more event sources by instantiating one or more event source objects in your code. Event sources act as conduits through which your application can raise events to the outside world. Any EIF events that you raise within your application must be raised through an event source. If you wish, you can use the default **Application** event source object that is provided for all instrumented applications.

Once the event sources are defined, then you need to add code to generate events for these sources. When raising an event in your code, you can specify the event source through which it should be raised. If you do not specify an event source when raising an event, it will be raised through the default **Application** event source.

Configuring EIF Instrumentation

Once your application has been instrumented, you need to create an initial EIF configuration that routes the events to event sinks. An application's EIF configuration is controlled by the EIF configuration file, which is called EnterpriseInstrumentation.config. This is an XML file that contains a number of XML configuration elements. The five main elements of the EIF configuration file are: **eventSource**, **eventSink**, **eventCategory**, **filter**, and **filterBindings**, and are explained below:

- One **eventSource** element is needed for each event source that is defined in the application.
- An **eventSink** element allows you to define groups of events based on their object type. Events can then be filtered to event sinks using the event categories.

- An **eventCategory** element allows you to define groups of events based on their object type. Events can then be filtered to event sinks according to these categories.
- A **filter** element connects event categories to event sinks.
- The **filterbinding** elements are used to connect event sources to event filters.

You must first generate the EIF configuration file and then edit it you suit your needs. You can use the InstallUtil.exe utility, which is part of the .NET Framework SDK, to generate a default EIF configuration file as a starting point. The file generated by this tool is not a complete configuration file. Next you need to complete the definition of the default filter elements in order to map events to event sinks. You can also add additional filters that allow you to select which events are processed and how they are processed.

The EIF configuration file can be edited manually or with a scriptable configuration API provided by EIF. Once an initial configuration file has been created, it is recommended that you develop scripts that perform common configuration tasks or changes. For example, one script might enable a filter that sends events to a Windows Event Trace sink, while another script might turn this filter off. Developers and operations staff should work together to determine the set of tasks that should be handled by configuration scripts.

Choosing Event Sinks

EIF provides built-in event sinks for WMI, the Windows Event Log service, and Windows Event Trace. Additionally, by implementing custom event sinks, events can be passed to other mechanisms. For example, application developers can develop custom event sinks to support logging events to a database.

The choice of the event sink(s) that you use is based on the monitoring requirements. These are typically determined by the operations team responsible for the deployed application. You are not limited to a single event sink but can use any combination of the three different types. Here are some things to consider when deciding which type of event sink should handle which type of event:

- The Trace Event Sink provides the best performance but has limited tools for reporting and analyzing the logs it generates. It is best suited for high-volume events that may generate hundreds of events per second.
- The WMI Event Sink is the richest eventing mechanism, with robust publish/subscribe support for consumers. Although WMI is not normally suitable for high-frequency eventing on Windows 2000, it is significantly more efficient on Windows Server 2003. WMI has a rich set of management tools.
- The Log Event Sink is suited for lower-volume events with a medium level of visibility. Examples are high-level audit messages and application warnings. The Windows Event Log service does not offer a highly structured format like WMI.

Request Tracing

Request Tracing is a key feature of EIF that allows you to trace business processes and application services by following an execution path across the tiers of a distributed .NET application. This path is defined by start and end points that have been inserted into the application's code by the developer. Any events that occur along this execution path are tagged with context-related fields that identify them as part of a particular request.

The Request Event Source is the conduit for the entire business process or service. It is also the control point for turning tracing on and off for that process.

Logging

If you want to use logging instead of EIF to store information about exceptions, then there are three possible options:

- The Windows Event Log service
- A central relational database, such as Microsoft SQL Server™ 2000
- A custom log file

Using the Windows Event Log

The Windows 2000 and Windows NT® operating systems provide a set of event logs. For example, the operating system places error details in the system event log, while an application event log is provided for applications to use. You can also isolate your application's log entries by creating a separate event log, rather than using the shared application event log. The event log provides a consistent and central storage repository for error, warning, and informational messages on a single machine. The .NET Framework provides event log related classes, making it easy to log and monitor.

Advantages

- It is a highly reliable method of logging information.
- It is provided by both Windows 2000 and Windows NT.
- It features log file size management. You can configure a maximum log size and specify the number of days for which records are to be maintained or whether log files are to be cleared manually.
- Tools such as the Event Viewer allow you to view and manipulate log entries.
- The .NET Framework classes, such as the **EventLog** class within the **System.Diagnostics** namespace, make the event log easy to write to and maintain programmatically.
- Most monitoring tools support the event log as an event source.
- The event log has tight integration with Windows Management Instrumentation.

- Systems administrators and operators are generally familiar with the event log and its associated tools.
- The operating system and many applications log to the event log, which makes it easy to compare the sequence of system and application events.

Disadvantages

- Each machine logs to a local event log. This can cause monitoring and maintenance problems for large server farms. You can write entries to a log on another machine; however, this increases the risk of failure and should be avoided.

Note that Microsoft Application Center Server, along with several third-party software packages, allows you to centralize the local event logs from multiple servers. This allows you to view multiple event logs as a single entity.

For more information on using the Windows Event Log service, see the following Web sites:

- Logging Application, Server, and Security Events
- Application Center Server homepage

Using a Central Relational Database

To solve the problem that the single-machine nature of the event log can cause, you can use a centralized database, such as SQL Server 2000, to store information.

Advantages

- All information is stored in a single centralized location and is accessible from remote machines.
- You can use database tools for querying and reporting the error data.
- You can structure the storage of the information in application specific tables.

Disadvantages

- Database logging introduces an element of risk. If your system is unable to access the database, perhaps due to a network failure, you run the risk of losing the information. To guarantee that your exception is logged, you may decide to write information to the local event log if the database update fails. This is not an ideal solution because you are then forced to monitor two locations and deal with two distinct error formats.
- Your application entries will be stored in a different location than the operating system's entries. This can make it difficult to compare the sequence of events between your applications logs and the operating system's logs.
- You must develop the tools to view and administer the data.
- You must customize monitoring systems to interact with the database.

Using a Custom Log File

Some applications log information to a custom log file. This solution should only be used in scenarios where the Event Log cannot satisfy some specific logging need.

Advantages

You have complete flexibility when choosing the log format.

Disadvantages

- It is not trivial to build a custom logging mechanism than can handle concurrent access.
- You must create and implement processes to manage the log size.
- You must develop the tools to view and administer the log files.
- This can cause monitoring and maintenance problems for large server farms since many monitoring tools, including Microsoft Application Center Server, cannot aggregate custom log files.
- Monitoring tools must be configured to monitor the log files.

Using Asynchronous Delivery with Message Queuing

In conjunction with the options above, you can use Microsoft Message Queuing to provide a reliable transport mechanism to deliver data asynchronously to a data store. While Message Queuing does not represent the final data store, it can be used as a reliable delivery mechanism to ensure that the logged data reaches its ultimate storage repository, which may be any of the locations discussed above. The transactional nature of Message Queuing ensures that log data will not be lost, even in the event of a server or network failure.

- Message Queuing represents an excellent choice of transport where you are striving for a centralized logging solution. By taking advantage of transactional Message Queuing queues, you can ensure that data will not be lost or duplicated and will be delivered reliably to the destination system.
- Message Queuing requires a larger coding effort. The code to convert the Message Queuing messages into the final store must be developed.
- Since Message Queuing is an asynchronous mechanism, the messages may not arrive to the final store in the order in which they occurred.
- Data may not be stored immediately, particularly if Message Queuing is configured to route messages through several queues. This may be an issue in situations where information is required immediately.

Notification

If you are using logging instead of EIF to keep track of exceptions, then you must also develop a notification solution. Notification is a critical component of any exception management system. While logging is essential in helping you understand what went wrong and what needs to be done to rectify the problem, notification informs you about the condition in the first place. Without proper notification, exceptions can go undetected.

The notification processes adopted by your application should be decoupled from your application code. You should not have to change code every time you need to modify your notification mechanism, for example to alter the list of recipients. Your application should communicate error conditions and rely on a decoupled monitoring system to identify those errors and take the appropriate action.

If your application is not going to run in an environment that utilizes a monitoring system, you have several options for creating notifications from your application. Either way, you need to work closely with any operations personnel or the monitoring system developers to define the correct procedures for monitoring and sending notifications.

Using a Monitoring Application

When working in an environment with a monitoring system, you can provide notifications in two ways.

- Your application can rely on the monitoring system to watch your log and deal with the detected error conditions appropriately.
- You can implement code to proactively raise events to notify the monitoring system. The preferred method for proactively notifying a monitoring system of a certain condition is to use WMI.

Monitoring the Log Data Store

You can configure monitoring systems to watch your log and monitor the arrival of messages. You may further fine tune the monitor to react only to a specific set of messages, or perhaps only to those that match a certain criteria, for example those that exceed a defined error level threshold. Tools such as Application Center Server can detect certain information logged to the event log and then take action based on preconfigured settings.

Advantages

- Your monitoring system is decoupled from your application code.
- You do not require any additional code beyond your logging implementation.

Disadvantages

- The monitoring system must be configured to poll the log and filter records to find those it needs to handle.
- Any other systems that want to handle these events must be configured to watch your chosen log.

For more information, see the Application Center Server.

Creating WMI Events

WMI is the Microsoft implementation of an industry standard for accessing management information in an enterprise environment. The WMI event infrastructure provides the ability for applications to raise WMI events that can be monitored and handled. This approach decouples the notification rules and procedures from the code in your application. Your application needs to raise only certain events that a monitoring system can catch, and then implement the correct notification procedures. .NET provides classes within the **System.Management** and **System.Management.Instrumentation** namespaces to simplify WMI programming.

You should not raise a WMI event for every exception. WMI events provide a hook into your application to allow other systems to react. They should therefore be used when something occurs within your application that could warrant action from an external process or application.

Advantages

- Using the WMI event infrastructure decouples the operational procedures carried out as a result of an unhandled exception from your application code.
- The WMI event infrastructure integrates well with monitoring applications.
- The WMI event infrastructure allows monitoring applications to group your application events with other application events to provide an enterprise-wide notification system.

Disadvantages

- An application may raise WMI events from any number of machines comprising the application. In most cases, a monitoring tool must be installed on each of the machines it needs to monitor in order to detect these events. Product licensing may become an issue in a distributed system.

For more information on WMI and creating WMI events, see the following resources:

- Monitoring in .NET Distributed Application Design
- Windows Management Instrumentation (WMI) SDK

Creating Notifications

If your application does not have the advantage of working with a monitoring tool, there are several ways to create notifications. Some common options include:

- Sending mail using SMTP
- Developing a custom notification system.

Sending Mail Using SMTP

One approach to sending notifications is to send mail to a distribution list that includes the appropriate support personnel. This approach is common because it is easy to implement, but is not the most robust or stable choice.

Advantages

- A solution is quick and easy to implement using SMTP. The .NET Framework provides the **System.Web.Mail** namespace to simplify the generation and transmission of mail.

Disadvantages

- This is not a resilient solution. If the e-mail fails, the notification will is lost.
- When the number of errors is high, you can generate many e-mail notifications, which makes it difficult to solve the problem at hand.

Developing a Custom Notification System

You can also create a notification program that can receive messages from your application and then take some configurable action based on the type of the exception. Message Queuing provides a reliable asynchronous delivery mechanism to pass data to a notification system (in the same way that it can be employed to transfer data to a centralized error log). By taking advantage of transactional Message Queuing queues, you can ensure that data is not lost or duplicated and is delivered reliably to the destination system. The custom notification application only needs to provide a subset of the functionality of a monitoring system.

Advantages

- Using a custom notification program that uses transactional Message Queuing queues decouples the notification procedures from your application code.
- A custom notification program that uses transactional Message Queuing queues can support multiple applications.
- A custom notification program eliminates potential licensing issues associated with a monitoring system.
- Using a custom notification program that uses transactional Message Queuing queues provides asynchronous, reliable delivery of notifications.

Disadvantages

- You must develop and maintain the notification system.
- Similar functionality is already provided in production monitoring systems such as Application Center Server.

Specific Technology Considerations

This section examines some specific considerations for exception management when combined with certain .NET technologies.

Interoperability

The runtime provides a variety of interoperability features that allow your managed and unmanaged code to work together. This also applies to exception handling. When a COM client calls a managed class that throws an exception, the runtime catches and translates it into an HRESULT that can be understood by the unmanaged COM code. Similarly, if a COM component returns a failed HRESULT to managed code, the runtime translates an exception from the HRESULT and incorporates any additional information provided by the COM **IErrorInfo** interface into the exception object. If the runtime does not recognize the HRESULT, it returns a generic **COMException** object with its **ErrorCode** property set to the unrecognized HRESULT value.

Many of the .NET Framework class library exceptions return new HRESULT values to COM components. This causes existing COM code not to recognize the HRESULTs. If your COM code takes some action based on certain failed HRESULTs, you should change the HRESULT value of the exception before propagating it to your COM code. The **HResult** property of the exception object represents the HRESULT value returned to unmanaged code. It is a read/write property and can be set to any value to ensure the correct operation of your COM code. In addition, any application exceptions you create that interact with unmanaged COM code should override the **HResult** property and default the value to an HRESULT value that accurately represents the exception condition.

For more information on interoperability, refer to the following:

- Microsoft .NET/COM Migration and Interoperability
- Handling COM Interop Exceptions
- HRESULTS and Exceptions (includes table of HRESULT to Exception mappings)

Localization

Localization is required to provide the right information to the right audience. Your application should provide localization to ensure that your information is presented in a format that your audiences can understand. .NET provides mechanisms, such as assembly cultures and satellite assemblies, to integrate localization features into your application.

- **Cultures**

  A culture is part of the assembly identity and is involved in the binding process. For example, an assembly could be strictly developed for a Spanish audience.

- **Satellite Assemblies**

  These are resource-only assemblies that do not contain common intermediate language (CIL) code. Satellite assemblies can specify a culture that accurately reflects the culture of the resources placed in the assembly.

For more information on localization, see the following:

- Deploying Resource Files
- Localization Sample
- Resource File Generator Utility
- .NET Framework: Building, Packaging, Deploying, and Administering Applications and Types, Part 2

Authors

Kenny Jones, Edward Jezierski, Jason Hogg, Roberta Leibovitz (Modeled Computation) and Colin Campbell (Modeled Computation)

Collaborators

Many thanks to the following contributors and reviewers:

Anders Hejlsberg, Jeffrey Richter (Wintellect), Rob Howard, Susan Warren, Bart Robertson, Edward Jezierski, Alex Mackman (Content Master Ltd.), Tony Surma, Chris Brooks, Brad Abrams, Paul Bates, David Keogh, Jayesh Rege, Ann Chung, Ken Argo, Bernard Chen (Sapient), Steve Busby, Jeff Kercher, Amitabh Srivastava, Peter Laudati.

To learn more about .NET best practices, please visit the patterns & practices Web page.

To participate in an online collaborative development environment on this topic, join the GotDotNet workspace: Microsoft Patterns & Practices Exception Management & Instrumentation Workspace. Please share your Exception Management Block questions, suggestions, and customizations with the community in this workspace.

Questions? Comments? Suggestions? For feedback on the content of this article, please e-mail us at devfdbck@microsoft.com.

Microsoft®

patterns & practices

Proven practices for predictable results

Patterns & practices are Microsoft's recommendations for architects, software developers, and IT professionals responsible for delivering and managing enterprise systems on the Microsoft platform. Patterns & practices are available for both IT infrastructure and software development topics.

Patterns & practices are based on real-world experiences that go far beyond white papers to help enterprise IT pros and developers quickly deliver sound solutions. This technical guidance is reviewed and approved by Microsoft engineering teams, consultants, Product Support Services, and by partners and customers. Organizations around the world have used patterns & practices to:

Reduce project cost

- Exploit Microsoft's engineering efforts to save time and money on projects
- Follow Microsoft's recommendations to lower project risks and achieve predictable outcomes

Increase confidence in solutions

- Build solutions on Microsoft's proven recommendations for total confidence and predictable results
- Provide guidance that is thoroughly tested and supported by PSS, not just samples, but production quality recommendations and code

Deliver strategic IT advantage

- Gain practical advice for solving business and IT problems today, while preparing companies to take full advantage of future Microsoft technologies.

To learn more about *patterns & practices* visit: *msdn.microsoft.com/practices*

To purchase *patterns & practices* guides visit: *shop.microsoft.com/practices*

patterns & practices
Proven practices for predictable results

patterns & practices

Proven practices for predictable results

Patterns & practices are available for both IT infrastructure and software development topics. There are four types of patterns & practices available:

Reference Architectures

Reference Architectures are IT system-level architectures that address the business requirements, operational requirements, and technical constraints for commonly occurring scenarios. Reference Architectures focus on planning the architecture of IT systems and are most useful for architects.

Reference Building Blocks

References Building Blocks are re-usable sub-systems designs that address common technical challenges across a wide range of scenarios. Many include tested reference implementations to accelerate development.

Reference Building Blocks focus on the design and implementation of sub-systems and are most useful for designers and implementors.

Operational Practices

Operational Practices provide guidance for deploying and managing solutions in a production environment and are based on the Microsoft Operations Framework. Operational Practices focus on critical tasks and procedures and are most useful for production support personnel.

Patterns

Patterns are documented proven practices that enable re-use of experience gained from solving similar problems in the past. Patterns are useful to anyone responsible for determining the approach to architecture, design, implementation, or operations problems.

To learn more about *patterns & practices* visit: *msdn.microsoft.com/practices*

To purchase *patterns & practices* guides visit: *shop.microsoft.com/practices*

patterns & practices current titles

December 2002

Reference Architectures

Microsoft Systems Architecture—Enterprise Data Center *2007 pages*
Microsoft Systems Architecture—Internet Data Center *397 pages*
Application Architecture for .NET: Designing Applications and Services *127 pages*
Microsoft SQL Server 2000 High Availability Series: Volume 1: Planning *92 pages*
Microsoft SQL Server 2000 High Availability Series: Volume 2: Deployment *128 pages*
Enterprise Notification Reference Architecture for Exchange 2000 Server *224 pages*
Microsoft Content Integration Pack for Content Management Server 2001 and SharePoint Portal Server 2001 *124 pages*
UNIX Application Migration Guide *694 pages*
Microsoft Active Directory Branch Office Guide: Volume 1: Planning *88 pages*
Microsoft Active Directory Branch Office Series Volume 2: Deployment and Operations *195 pages*
Microsoft Exchange 2000 Server Hosting Series Volume 1: Planning *227 pages*
Microsoft Exchange 2000 Server Hosting Series Volume 2: Deployment *135 pages*
Microsoft Exchange 2000 Server Upgrade Series Volume 1: Planning *306 pages*
Microsoft Exchange 2000 Server Upgrade Series Volume 2: Deployment *166 pages*

Reference Building Blocks

Data Access Application Block for .NET *279 pages*
.NET Data Access Architecture Guide *60 pages*
Designing Data Tier Components and Passing Data Through Tiers *70 pages*
Exception Management Application Block for .NET *307 pages*
Exception Management in .NET *35 pages*
Monitoring in .NET Distributed Application Design *40 pages*
Microsoft .NET/COM Migration and Interoperability *35 pages*
Production Debugging for .NET-Connected Applications *176 pages*
Authentication in ASP.NET: .NET Security Guidance *58 pages*
Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication *608 pages*

Operational Practices

Security Operations Guide for Exchange 2000 Server *136 pages*
Security Operations for Microsoft Windows 2000 Server *188 pages*
Microsoft Exchange 2000 Server Operations Guide *113 pages*
Microsoft SQL Server 2000 Operations Guide *170 pages*
Deploying .NET Applications: Lifecycle Guide *142 pages*
Team Development with Visual Studio .NET and Visual SourceSafe *74 pages*
Backup and Restore for Internet Data Center *294 pages*

For current list of titles visit: *msdn.microsoft.com/practices*

To purchase *patterns & practices* guides visit: *shop.microsoft.com/practices*

patterns & practices
Proven practices for predictable results