developer.*
The Independent Source for Software Professionals

.NET Exception Handling

by Edward G. Nilges

Proper error handling is the bane of the developer's existence, for errors are both visible and marginal. Errors are visible in that the user sees them. They are marginal in that they are part of "infrastructure;" error handling is a technical problem, and not a business priority.

The marginality of error handling means that the "business case" for error handling is weak, since a heavy duty error handling scheme is to a manager with an MBA from a good school, a market failure.

Unlike programmers, people who are fiduciaries of companies—that is, people with direct responsibility to the bottom line—generally seek ways to quantify risk, and this is usually using an insurance or hedging model. They don't like to see elaborate error handling in cases where this cost hasn't been externalized into a customer demand for elaborate error handling or even a software risk insurance policy which states, "we underwrite on condition your programmers do thus and so in error handling."

But this externalization of costs leads to a further problem: as soon as the business requirements get spelled out in detail the question arises to why they need to be in effect rewritten as code. This, in turn, takes us back to the central issue of programming: its necessary marginality as a form of writing, supplementary to the specification.

But that is a large theme; let us focus on error handling.

In my view, a system-level error handling scheme in .NET must meet the following criteria:

- To leave error logging, notification, and display to the highest level tier.
- To avoid elaborate recursive error handling of errors within the error handling, a tricky and in some cases desperate business.
- To add value to the error message where possible, and to Throw or otherwise pass the error message to the next higher level of error handling. To never "lock down" the code to any one vision of error handling.

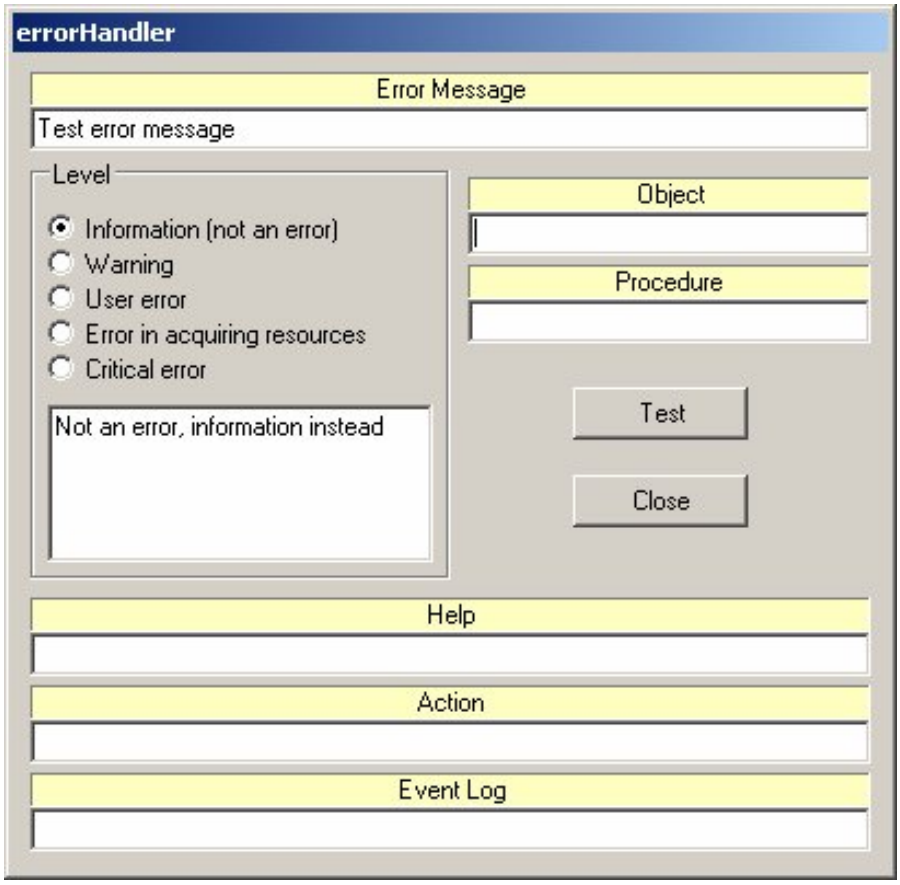
- To add value in part by conscientiously decorating the message with highly structured information, including the date, the time, the class from where the error emerged, the object instance from where the error emerged, and the type of the error (see below).
- To provide at least the option to write the decorated error to a system log.
- To provide help in two forms
    - Help help (general information)
    - Action help (information as to what the object is going to do about the error)
- To force the error caller to think about the type of error:
    - Pure information that is not, in fact, an error, but which "zero case" the error handles as a professional courtesy...while making sure the error viewer knows that the "error" is not an error
    - Warnings, veiled threats, and words to the wise: any situation in which processing can continue without error and in which the object issuing the error hasn't been compromised or damaged
    - Serious errors: processing cannot continue without error. These seem to consist of
        - User errors, which can be recovered from in the sense they don't affect the integrity of the object
        - Resource errors, which while not compromising the integrity of the object, prevent it from doing its job in responding to current and future events
    - Critical errors: errors where the software discovers a program bug. For example, the default "not supposed to happen" case in a select case statement should raise this type of error. The object should put itself out of commission.

I have created a demonstration project that illustrates these principles in a .NET-based exception handling framework. After a walk-through of the demonstration/test program I will conclude with an explanation of the underlying code. The entire project, including the test program and a reusable .NET error handling DLL, can be downloaded from this URI:

```
http://www.developerdotstar.com/nilges/errorHandler.zip
```

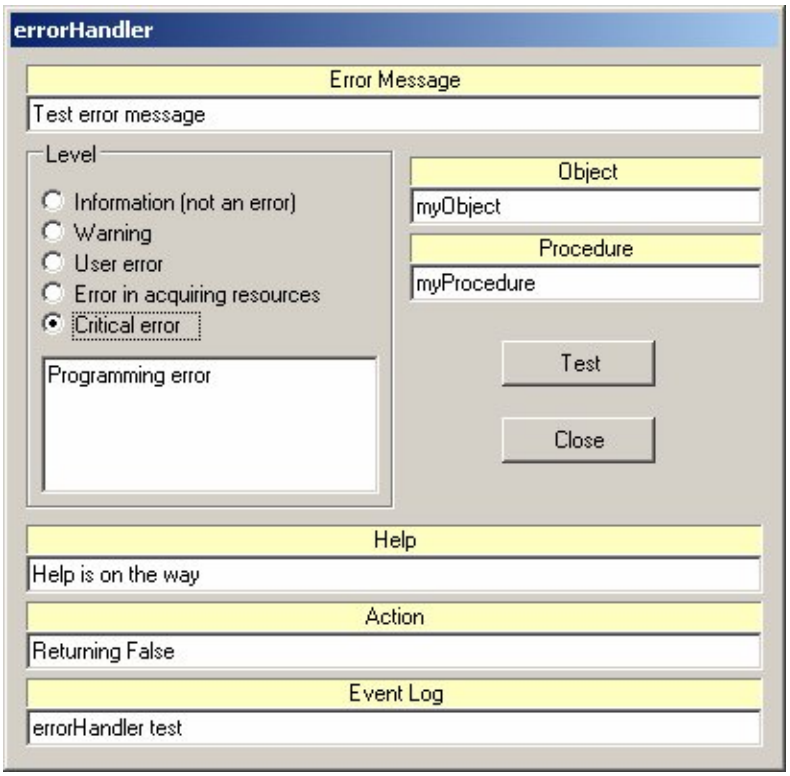
Trying the errorHandler out

In the shipped code itemized in the next section, run `errorHandlerTest.EXE` to see the following screen:



In the Level group box, click all the radio buttons to see the result of the `errorLevel2Explanation()` procedure (see documentation later in this article) refresh the screen with the explanation of each level.

When you're done exploring, leave the level at "Critical error." Fill in values as follows:



Click Test. Since `errorHandlerTest` places the call to `errorHandler` in a `Try..Catch` block, the `Catch` intercepts the error that `errorHandler` Throws to the next higher level along with a vanilla Exception. Since `errorHandlerTest` is the highest level of error handling and it "knows" it is a Windows application it makes sense to allow `errorHandlerTest` to report the error using a Windows message box. Here is what should appear:

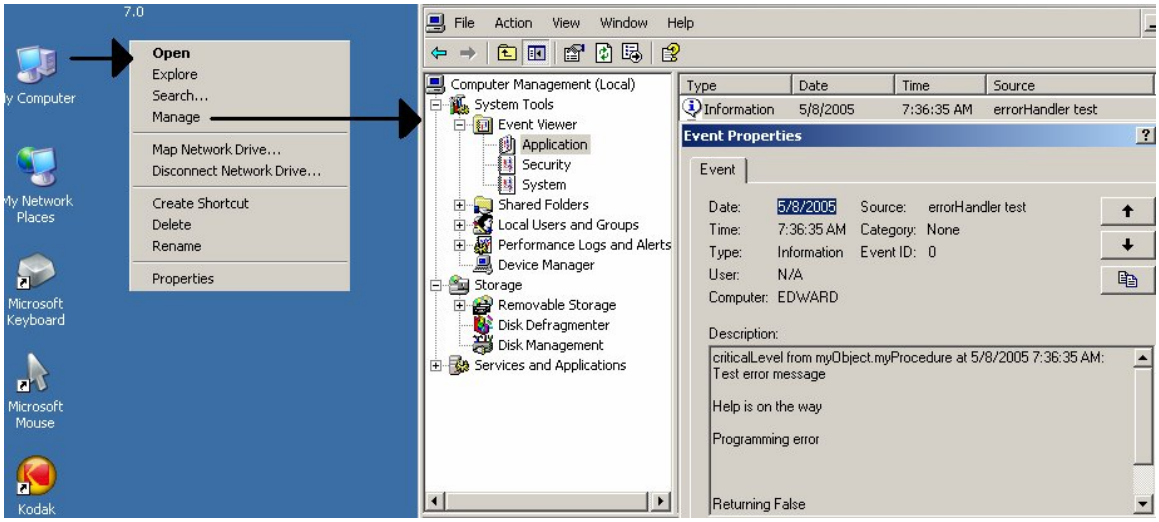
The complete error message consists of several lines:

- The first line starts with `System.Exception` because the top level test form has used `Catch objException As Exception` to Catch the error, and has simply displayed `objException.ToString()`: but because correspondingly, `errorHandler` created a New exception with a string set to the full error message, the text of the `Exception.ToString()` is what `errorHandler` has built
- Therefore, `System.Exception` is followed by the meaning of the radio button selected as the Level of the error: `criticalLevel`.
- This is followed by the source of the error: `myObject.myProcedure()`.
- This is followed by the date, time, and the body of the error message.
- A line break is followed by the full help advice which consists of help text, the `errorLevel2Explanation` explanation of the error, a blank line, and finally the action you identified.

  The blank line appears because `errorHandler` accesses `Microsoft.VisualBasic.Err.Number` and `Err.Description`. If `Err.Number` is zero and `Err.Description` is a null string, a blank line appears, otherwise these values are displayed.

  You may want to remove the code that displays `Err.Number` and `Err.Description`. This is because using them increases your dependence on `Microsoft.VisualBasic` as a library. This library is full of "legacy" code which doesn't in all cases work smoothly in Web Services and as a general rule, `VB.NET` code should migrate away from it.
- The help advice lines are followed by the rest of the `Exception.ToString()` text, which provide the source of the error.

Take a look at the Event log. Right click My Computer on the Windows desktop, click Manage, open System Tools and Event Viewer if necessary under Computer Management, and open the most recent Application event: this will have the Source set to errorHandler Test:



As shown above, because `strEventLog` was included, an event was added to the Application log.

Try the `errorHandler` out to see if it meets your needs and that of your team.

Additional Code Explanation

Project Structure

The following directories and files are available, containing the `errorHandler` and its test application.

- ErrorHandler: directory containing all files needed
- ErrorHandler/AssemblyInfo.vb: assembly information for `errorHandler.DLL`
- ErrorHandler.SLN: solution file including both `errorHandler` and `errorHandlerTest`
- ErrorHandler.PROJ: project file for `errorHandler.DLL`
- ErrorHandler.VB: Visual Basic .Net source code for `errorHandler.DLL`
- Bin: directory containing `errorHandler.DLL`
- ErrorHandlerTest: directory containing the errorHandler test application, containing `ErrorHandlerTest.SLN`, `ErrorHandlerTest.PROJ`, and `Form1.VB`.

The errorHandler Class

`errorHandler` is a stateless .NET class which exposes two Shared methods: `errorHandler()` and `errorLevel2Explanation()`:

errorHandler()

```
errorHandler(strMessage As String,
             ByVal enuLevel As ENUerrorHandlerLevel,
             ByVal strObject As String,
             ByVal strProcedure As String,
             ByVal strHelp As String,
             ByVal strAction As String,
             ByVal strEventLog As String)
```

`strMessage` is the only required parameter of `errorHandler()` and it should describe the message. There's an art to composing an error message, of course, and `errorHandler()` can't prevent you from composing a bad one. In general the message should avoid blaming the user who unlike the programmer is usually scared of error messages.

The remaining parameters are all optional in the sense of overloads-optional.

- enuLevel is the error level as described above.
- criticalLevel: a programming error has occurred.
- informationLevel: not an error: but we need to provide some information using the error handling system.
- resourceLevel: an object cannot get the resources it needs from the system to operate properly. In .Net, for example, this would be any failure to create a reference object (other than Boolean, Byte, Short, Integer, Long, Single, Double or String), which needs storage in the .Net heap.
- userLevel: the object user has supplied invalid data, typically as a procedure parameter, such that the request made cannot be honored, but the object itself is OK.
- warningLevel: there might be a problem, but probably isn't.

strObject should identify the object producing the error...not its class. Ideally, you've named each class instance using a method I will fully describe in a subsequent article, where the name includes the class name, a sequence number, and the date and time. However, something like *classname* instance is perfectly acceptable for short-term use. It identifies the class, but makes it clear that an *instance* produced the error. `strObject` defaults to "Unidentified object."

strProcedure should identify the "procedure" which in VB.Net identifies either the method or property, generating the error. `strProcedure` defaults to "Unidentified procedure."

strHelp can be a long or short string with long or brief help information. There doesn't seem to be much point in subordinating an elaborate "help" system to the errorHandler which to be reusable interface needs to be light in weight and in spirit, and you can, if you like, get the help information from another subsystem and pass it, if it is of reasonable size, as a string. `strHelp` defaults to a null string because it looks unprofessional to default to "Help is not available." There's a public service ad on Hong Kong TV which advises people working in shops never to say "no" in such a way as to make the customer feel as if she's undeserving of help.

strAction should identify what is being done in response to the error condition. It's a separate consideration from help information. It defaults to a null string.

strEventLog when present and not null causes the error to be logged (with its full "decoration" including date and time) using `System.Diagnostics.EventLog.WriteEntry()` with two parameters: `strEventLog` and the fully decorated message.

On a Windows system, this adds the message with decoration and new-lines to the Application log which can be viewed by right-clicking My Computer on the desktop, opening Manage, opening System Tools and Event Viewer and clicking Application.

errorLevel2Explanation()

This shared procedure accepts an `enuLevel` enumerator (of type ENUerrorHandlerLevel) and returns its explanation:

- criticalLevel: "Programming error" is returned
- informationLevel: "Not an error, information instead" is returned
- resourceLevel: "System not able to get a needed resource" is returned
- warningLevel: "Warning, processing continues: there may be a problem" is returned
- userLevel: "Have recovered from a user error" is returned

One problem throughout this short and simple class is its exclusive use of English, but for ease of modification, the error level explanations are symbol constants, and not strings.

###

Edward G. Nilges, after thirty years as a software developer in which he debugged a Fortran compiler in object code and assisted the real-life protagonist of the film *A Beautiful Mind* at Princeton, is now a free programmer of open source and a teacher of computer science, English, and philosophy in Hong Kong. He lives on Lamma Island, off the big island, where on dune and headland sink the fire.