Exception Handling Best Practices in .NET

Daniel Turini, 21 Feb 2005 CPOL

4.90 (406 votes)

Design guidelines for exception handling in .NET which will help you to create more robust software

Contents

Introduction

"My software never fails". Can you believe it? I'm almost hearing you all, screaming that I'm a liar. "Software that never fails is something near to impossible!"

Contrary to common belief, creating reliable, robust software is not something near to impossible. Notice that I'm

not referring to bug-free software, intended to control nuclear power plants. I'm referring to common business software, which can run unattended on a server, or even a desktop machine, for long periods of time (weeks or months) and work predictably without any significant glitch. By predictably, I mean that it has a low failure rate, you can easily understand failure conditions to fix it quickly, and it never damages data in response of an external failure.

In other words, software that is stable.

Having a bug in your software is forgivable, and even expected. What's unforgivable is having a recurring bug you can't fix because you don't have enough information.

To understand better what I'm saying, I've seen countless business software that, in an out of disk space error in the DBMS, reports something like this:

"Could not update customer details. Contact the system administrator and try again later".

While this message may be an adequate of reporting an unknown resource failure to a business user, all too often this is the whole debugging information that is available to debug the error cause. Nothing was logged, and understanding what is happening will be time-consuming and often programmers will guess a lot of possible causes until they find the real error cause.

Notice that in this article, I will concentrate only in how to make a better use of .NET exceptions: I won't discuss how to properly report error messages, because I believe this belongs to UI domain, and it depends heavily on the interface being developed and the target audience; a blog text editor targeting teenagers should report error messages in a way completely different than a socket server, which will only be used directly by programmers.

Plan for the worst

A few basic design concepts will make your program much more robust, and will improve the user experience in the presence of an unexpected error. What do I mean by "improve the user experience in the presence of an unexpected error"? It's not that the user will be thrilled by the marvelous dialog box you'll show him. It's more about don't corrupting data, don't crashing the computer, and behaving safely. If your program can pass through an out of disk space error without doing any harm, you improved the user experience.

Check it early

Strong type checking and validation are powerful tools to prevent unexpected exceptions and to document and test code. The earlier in execution you detect a problem, the easier is to fix. Trying to understand what a `CustomerID` is doing on the `ProductID` column on the InvoiceItems table after a few months isn't fun neither easy. If you used classes for storing customer data, instead of using primitives (e.g., `int`, `string`, etc), chances are that the compiler would never allow you to do such a thing.

Don't trust external data

External data is not reliable. It must be extensively checked. It doesn't matter if the data is coming from the registry, database, from a disk, from a socket, from a file you just wrote or from the keyboard. All external data should be checked and only then you can rely on it. All too often I see programs that trust configuration files because the programmer never thought that someone would edit the file and corrupt it.

The only reliable devices are: the video, the mouse and keyboard.

Anytime you need external data, you can have the following situations:

- Not enough security privileges
- The information is not there

- The information is incomplete
- The information is complete, but invalid

It really doesn't matter if it's a registry key, a file, a socket, a database, a web service or a serial port. All external data sources will fail, sooner or later. Plan for a safe failure and minimize damage.

Writes can fail, too

Unreliable data sources are also unreliable data repositories. When you're saving data, similar situations can happen:

- Not enough security privileges
- The device isn't there
- There's not enough space
- The device has a physical fault

That's why compression programs create a temporary file and rename it after they're done, instead of changing the original one: if the disk (or even the software) fails for some reason, you won't lose your original data.

Code Safely

A friend of mine often says: "A good programmer is someone who never introduce bad code in his projects". I don't believe that this is all that it's needed to be a good programmer, but surely it will put you almost there. Below, I compiled a list of the most common "bad code" that you can introduce in your projects, when it comes to exception handling.

Don't throw new Exception()

Please, don't do it. `Exception` is a too broad class, and it's hard to catch without side-effects. Derive your own exception class, but derive it from`ApplicationException`. This way you could set a specialized exception handler for exceptions thrown by the framework and another for exceptions thrown by yourself.

Revision note: David Levitt wrote me, in the comments section below, that although Microsoft still touts using `System.ApplicationException` as a base class in MSDN docs, this is no longer considered a good practice, as noted by Brad Adams, as you can see in his blog. The idea is creating exception class hierarchies that are as shallow and wide as possible, as you often do with class hierarchies. The reason I didn't change the article immediately was because I needed to do more research before I introduced it here. After all this research, I could not decide yet whether shallow class hierarchies are a good idea in Exception handling or not, so I decided to leave both opinions here. But, no matter what you do, don't throw new `Exception()` and derive your own`Exception` class when needed.

Don't put important exception information on the Message field

Exceptions are classes. When you return the exception information, create fields to store data. If you fail on doing it, people will need to parse the Message field to get the information they need. Now, think what will happen to the calling code if you need to localize or even just correct a spelling error in error messages. You may never know how much code you'll break by doing it.

Put a single catch (Exception ex) per thread

Generic exception handling should be done in a central point in your application. Each thread needs a separate try/catch block, or you'll lose exceptions and you'll have problems hard to understand. When an application starts several threads to do some background processing, often you create a class for storing processing results. Don't forget to add a field for storing an exception that could happen or you won't be able to communicate it to the

main thread. In "fire and forget" situations, you probably will need to duplicate the main application exception handler on the thread handler.

Generic Exceptions caught should be published

It really doesn't matter what you use for logging - log4net, EIF, Event Log, TraceListeners, text files, etc. What's really important is: if you caught a generic Exception, log it somewhere. But log it only once - often code is ridden with catch blocks that log exceptions and you end up with a huge log, with too much repeated information to be useful.

Log Exception.ToString(); never log only Exception.Message!

As we're talking about logging, don't forget that you should always log`Exception.ToString()`, and never `Exception.Message`.`Exception.ToString()` will give you a stack trace, the inner exception and the message. Often, this information is priceless and if you only log`Exception.Message`, you'll only have something like "Object reference not set to an instance of an object".

Don't catch (Exception) more than once per thread

There are rare exceptions (no pun intended) to this rule. If you need to catch an exception, always use the most specific exception for the code you're writing.

I always see beginners thinking that good code is code that doesn't throw exceptions. This is not true. Good code throws exceptions as needed, and handles only the exceptions it knows how to handle.

As a sample of this rule, look at the following code. I bet that the guy who wrote it will kill me when he read this, but it was taken from a real-world example. Actually, the real-world code was a bit more complicated - I simplified it a lot for didactic reasons.

The first class (`MyClass`) is on an assembly, and the second class (`GenericLibrary`) is on another assembly, a library full of generic code. On the development machine the code ran right, but on the QA machines, the code always returned "Invalid number", even if the entered number was valid.

Can you say why this was happening?

```
public class MyClass
{
    public static string ValidateNumber(string userInput)
    {
        try
        {
            int val = GenericLibrary.ConvertToInt(userInput);
            return "Valid number";
        }
        catch (Exception)
        {
            return "Invalid number";
        }
    }
}

public class GenericLibrary
{
    public static int ConvertToInt(string userInput)
    {
        return Convert.ToInt32(userInput);
    }
}
```

The problem was the too generic exception handler. According to the MSDN documentation, `Convert.ToInt32` only

throws `ArgumentException`,`FormatException` and `OverflowException`. So, those are the only exceptions that should be handled.

The problem was on our setup, which didn't include the second assembly (`GenericLibrary`). Now, we had a `FileNotFoundException` when the`ConvertToInt` was called, and the code assumed that it was because the number was invalid.

The next time you write "`catch (Exception ex)`", try to describe how your code would behave when an `OutOfMemoryException` is thrown.

Don't ever swallow exceptions

The worst thing you can do is catch (Exception) and put an empty code block on it. Never do this.

Cleanup code should be put in finally blocks

Ideally, since you're not handling a lot of generic exceptions and have a central exception handler, your code should have a lot more finally blocks than catch blocks. Never do cleanup code, e.g., closing streams, restoring state (as the mouse cursor), outside of a finally block. Make it a habit.

One thing that people often overlook is how a try/finally block can make your code both more readable and more robust. It's a great tool for cleanup code.

As a sample, suppose you need to read some temporary information from a file and return it as a string. No matter what happens, you need to delete this file, because it's temporary. This kind of return & cleanup begs for a try/finally block.

Let's see the simplest possible code without using try/finally:

```
string ReadTempFile(string FileName)
{
    string fileContents;
    using (StreamReader sr = new StreamReader(FileName))
    {
        fileContents = sr.ReadToEnd();
    }
    File.Delete(FileName);
    return fileContents;
}
```

This code also has a problem when an exception is thrown on, e.g., the`ReadToEnd` method: it leaves a temporary file on the disk. So, I've actually saw some people trying to solve it coding as this:

```
string ReadTempFile(string FileName)
{
    try
    {
        string fileContents;
        using (StreamReader sr = new StreamReader(FileName))
        {
            fileContents = sr.ReadToEnd();
        }
        File.Delete(FileName);
        return fileContents;
    }
    catch (Exception)
    {
        File.Delete(FileName);
        throw;
    }
}
```

The code is becoming complex and it's starting to duplicate code.

Now, see how much cleaner and robust is the try/finally solution:

```
string ReadTempFile(string FileName)
{
    try
    {
        using (StreamReader sr = new StreamReader(FileName))
        {
            return sr.ReadToEnd();
        }
    }
    finally
    {
        File.Delete(FileName);
    }
}
```

Where did the fileContents variable go? It's not necessary anymore, because we can return the contents and the cleanup code executes after the return point. This is one of the advantages of having code that can run **after** the function returns: you can clean resources that may be needed for the return statement.

Use "using" everywhere

Simply calling `Dispose()` on an object is not enough. The `using` keyword will prevent resource leaks even on the presence of an exception.

Don't return special values on error conditions

There are lots of problems with special values:

- Exceptions makes the common case faster, because when you return special values from methods, each method return needs to be checked and this consumes at least one processor register more, leading to slower code
- Special values can, and will be ignored
- Special values don't carry stack traces, and rich error details
- All too often there's no suitable value to return from a function that can represent an error condition. What value would you return from the following function to represent a "division by zero" error?

```
public int divide(int x, int y)
{
    return x / y;
}
```

Don't use exceptions to indicate absence of a resource

Microsoft recommends that you should use return special values on extremely common situations. I know I just wrote the opposite and I also don't like it, but life is easier when most APIs are consistent, so I recommend you to adhere to this style with caution.

I looked at the .NET framework, and I noticed that the almost only APIs that use this style are the APIs that return some resource (e.g.,`Assembly.GetManifestStream` method). All those APIs return null in case of the absence of some resource.

Don't use exception handling as means of returning information from a method

This is a bad design. Not only exceptions are slow (as the name implies, they're meant only to be used on exceptional cases), but a lot of try/catch blocks in your code makes the code harder to follow. Proper class design can accommodate common return values. If you're really in need to return data as an exception, probably your method is doing too much and needs to be split.

Use exceptions for errors that should not be ignored

I'll use a real world example for this. When developing an API so people could access Crivo (my product), the first thing that you should do is calling the `Login`method. If `Login` fails, or is not called, every other method call will fail. I chose to throw an exception from the Login method if it fails, instead of simply returning false, so the calling program cannot ignore it.

Don't clear the stack trace when re-throwing an exception

The stack trace is one of the most useful information that an exception carries. Often, we need to put some exception handling on catch blocks (e.g., to rollback a transaction) and re-throw the exception. See the right (and the wrong) way of doing it: The wrong way:

```
try
{
    // Some code that throws an exception
}
catch (Exception ex)
{
    // some code that handles the exception
    throw ex;
}
```

Why is this wrong? Because, when you examine the stack trace, the point of the exception will be the line of the "`throw ex;`", hiding the real error location. Try it.

```
try
{
    // Some code that throws an exception
}
catch (Exception ex)
{
    // some code that handles the exception
    throw;
}
```

What has changed? Instead of "`throw ex;`", which will throw a new exception and clear the stack trace, we have simply "`throw;`". If you don't specify the exception, the throw statement will simply rethrow the very same exception the catch statement caught. This will keep your stack trace intact, but still allows you to put code in your catch blocks.

Avoid changing exceptions without adding semantic value

Only change an exception if you need to add some semantic value to it - e.g., you're doing a DBMS connection driver, so the user doesn't care about the specific socket error and wants only to know that the connection failed.

If you ever need to do it, please, keep the original exception on the InnerException member. Don't forget that your exception handling code may have a bug too, and if you have InnerException, you may be able to find it easier.

Exceptions should be marked [Serializable]

A lot of scenarios needs that exceptions are serializable. When deriving from another exception class, don't forget to add that attribute. You'll never know when your method will be called from Remoting components or Web Services.

When in doubt, don't Assert, throw an Exception

Don't forget that `Debug.Assert` is removed from release code. When checking and doing validation, it's often

better to throw an Exception than to put an assertion in your code.

Save assertions for unit tests, for internal loop invariants, and for checks that should never fail due to runtime conditions (a very rare situation, if you think about it).

Each exception class should have at least the three original constructors

Doing it is easy (just copy & paste the definitions from other exception classes) and failing to do that won't allow users of your classes to follow some of these guidelines.

Which constructors I am referring to? The last three constructors described onthis page.

Be careful when using the AppDomain.UnhandledException event

Revision note: I was pointed by Phillip Haack in my blog of this important omission. Other common source of mistakes is the`Application.ThreadException` event. There are lots of caveats when using them:

- The exception notification occurs too late: when you receive the notification your application will not be able to respond to the exception anymore.
- The application will finish if the exception occurred on the main thread (actually, any thread that started from unmanaged code).
- It's hard to create generic code that works consistently. Quoting MSDN:"*This event occurs only for the application domain that is created by the system when an application is started. If an application creates additional application domains, specifying a delegate for this event in those applications domains has no effect.*"
- When the code is running those events, you won't have access to any useful information other than the exception itself. You won't be able to close database connections, rollback transactions, nor anything useful. For beginners, the temptation of using global variables will be huge.

  Indeed, you should never base your whole exception handling strategy on those events. Think of them as "safety nets", and log the exception for further examination. Later, be sure to correct the code that is not handling properly the exception.

Don't reinvent the wheel

There are lots of good frameworks and libraries to deal with exceptions. Two of them are provided by Microsoft and I mention here:

- Exception Management Application Block
- Microsoft Enterprise Instrumentation Framework

Notice, though, that if you don't follow strict design guidelines, like those I showed here, those libraries will be nearly useless.

VB.NET

If you read through this article, you'll notice that all the samples I used were in C#. That's because C# is my preferred language, and because VB.NET has a few guidelines of its own.

Emulate C# "using" statement

Unfortunately, VB.NET still doesn't have the `using` statement. Whidbey will have it, but until it's released, everytime you need to dispose an object, you should use the following pattern:

```
Dim sw As StreamWriter = Nothing
Try
    sw = New StreamWriter("C:\crivo.txt")
    ' Do something with sw
Finally
    If Not sw is Nothing Then
        sw.Dispose()
    End if
End Finally
```

If you're doing something different when calling `Dispose`, probably you're doing something wrong, and your code can fail and/or leak resources.

Don't use Unstructured Error Handling

Unstructured Error Handling is also known as `On Error Goto`. Prof. Djikstra did it very well in 1974 when he wrote "*Go To statement considered harmful*". It was more than 30 years ago! Please, remove all traces of Unstructured Error Handling from your application as soon as possible. `On Error Goto` statements will bite you, I'll assure you.

Conclusion

I hope that this article helps someone to code better. More than a closed list of practices, I hope that this article be a starting point for a discussion of how to deal with exceptions in our code, and how to make our programs more robust.

I can't believe that I wrote all of this without any mistake or controversial opinion. I'd love to hear your opinion and suggestions about this topic.

History

- 9 Feb 2005 - Initial version
- 21 Feb 2005 - Added information about `ApplicationException`,`AppDomain.UnhandledException` and`Application.ThreadException`

License

Share

About the Author

Daniel Turini

CEO

Brazil

I develop software since I was 11. In the past 20 years, I developed software and used very different machines and languages, since Z80 based ones (ZX81, MSX) to mainframe computers. I still have passion for ASM, though no use for it anymore.
Professionally, I developed systems for managing very large databases, mainly on Sybase and SQL Server. Most of the solutions I write are for the financial market, focused on credit systems.
To date, I learned about 20 computer languages. As the moment, I'm in love with C# and the .NET framework, although I only can say I'm very proficient at C#, VB.NET(I'm not proud of this), T/SQL, C++ and libraries like ATL and STL.
I hate doing user interfaces, whether Web based or not, and I'm quite good at doing server side work and reusable components.
I'm the technical architect and one of the authors of Crivo, the most successful automated credit and risk assessment system available in Brazil.