Debugging Strategies for .NET Developers

DARIN DILLON

Apress™

ISBN (pbk): 1-59059-059-7

Printed and bound in the United States of America 12345678910

Technical Reviewer: Jonathan Morrison

Editorial Directors: Dan Appleman, Gary Cornell, Jason Gilmore, Simon Hayes, Martin Streicher, Karen Watterson, John Zukowski

Managing and Production Editor: Grace Wong

Copy Editor: Ami Knox

Proofreader: Laura Cheu

Compositor: Susan Glinert Stevens

Illustrator: Tony Jonick

Indexer: Valerie Haynes Perry

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.

Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax: 510-549-5939, email info@apress.com, or visit http://www.apress.com.

CHAPTER 4

Assertion Debugging

Q: How do you keep a computer programmer in the shower forever?
A: Give him a bottle of shampoo with the directions, "Lather, rinse, repeat."

Debug asserts are a tough sell. Lots of developers have never heard of them, so they don't use them. Lots of other developers have heard of them, but don't understand them and use them incorrectly. And some developers have watched the second group misuse asserts for so long that they've decided asserts are evil and should never be used at all. That's unfortunate, because when used correctly, asserts are one of the most powerful and cost-effective tools in your arsenal for detecting bugs and pinpointing their source.

The great thing about debugging with asserts is that they do a large part of identifying the problem for you. Once you get the hang of them, you'll often be able to solve bugs by just dropping in a bunch of asserts in the relevant area of your code, and then waiting for an assert to fire the moment your code steps out of line. Then you'll know the exact line that the error first appears on. It's not always quite that easy, of course, but there's no denying that asserts are a tremendous aid. Most developers don't use asserts nearly as much as they should.

In this chapter, we'll discuss what asserts are, what they aren't, and how to use them to quickly identify bugs at the point of failure. We'll also examine the specific classes .NET provides for asserts, and how to customize those classes for situations where the standard assert won't work.

What Are Asserts?

The idea behind asserts is so very simple. When writing code, you realize some condition is always expected to be true: A certain variable will always be greater than zero, for instance. If that condition is ever not true, then there is a bug. Of course, your code should try to handle the unexpected condition as gracefully as possible, but don't you want your program to notify you when there's a bug? You want good error handling so that the end user never even notices the bug; but at the same time, you yourself want to be informed of all bugs so you can fix them, regardless of whether they are handled or not. So in addition to handling the error, you also give yourself a warning message:

```
if (age < 0) {
    MessageBox.Show("Error: age cannot be < 0!");
    //Now handle the error as best as you can
    ...
}
```

That way, if the error condition ever occurs, the program will halt until you acknowledge the problem. After all, the code is in a state you thought it could never be in—either because of a bug or else because you didn't anticipate a legitimate condition. Either way, this condition is worthy of your attention. You don't have to fix the program right now—you could just hit the button to dismiss the `MessageBox` and let the error handling code take care of it—but you do have to at least acknowledge, "Yes, I understand my program is doing something I thought it would never do, and at some point I need to fix that."

The Problems with Message Boxes

Except there's a big problem with using a message box to display that warning message. While developing the code, you want all errors to be highly visible so you can notice and fix them before the code ships. However, your attitude changes once the code ships. You don't want to worry the customer with warning messages on every bug, so now you *do* want your error handling code to hide all bugs. Therefore, before shipping, you'll have to go through your code and remove all of those `MessageBox.Show` functions. That's a lot of work, and odds are good you may forget one. There must be a better way.

So how do you show highly visible warning messages to the developer without also showing them to the end user? One option is to write the warning messages to a log instead of displaying message boxes on the screen. But while logging errors is a good practice for other reasons (see Chapter 5), it's not so good for this problem here. Warning messages in the log are easy to ignore. Do you carefully scrutinize every line of your program's log after every single test run of your program? Few developers do.

The best way to catch these errors is to show an in-your-face message that's impossible to miss, and errors in the log are just too easy to ignore (or at least convince yourself, "Oh that's not important—I'll deal with that later maybe."). Besides, you'll only notice the messages in your log after the program run is finished. It's better to pop up something the instant the bug occurs so you can jump in with the debugger to figure out what's going on.

Solution: If we want a message box to show up for the developer but not for the user, and if we don't want the nightmare of maintaining two different versions of the code, then we can use *conditional compilation* to produce both a "Debug" and

"Release" build. Unlike Java, .NET supports conditional compilation: the ability to run certain code only in Debug mode but not in Release mode. So we could write

```
#if DEBUG
    MessageBox.Show("Error: blah blah blah");
#endif
```

Now we have a single version of the code that can be compiled in two ways: The message box will appear in Debug mode, but not in Release mode. This gives us the best of both worlds: We only have to maintain one version of source code, but by changing a single switch in the compiler, we can produce a debug build for our own personal testing or a release build for formal testing by the quality assurance department and eventual release.

TIP Actually, .NET supports conditional compilation for any symbols you like, not just `DEBUG` and `RELEASE`. Just as in C and C++, you can define your own symbols: `USING_VERSION_1_GUI` and `USING_VERSION_2_GUI`, for instance. This feature can easily be misused, but it does allow great flexibility when you're not yet sure which direction the code is heading.

Writing `#if DEBUG ... #endif` over and over grows tedious. Instead, we can use a shortcut. The .NET library includes a great helper class called `System.Diagnostics.Debug`. Inside that class, we simply call the `Assert` method to get exactly what we want: The message box will appear in Debug mode, but not in Release mode.

```
System.Diagnostics.Debug.Assert(x >= 0);
```

Using Asserts to Launch the Debugger

In fact, that `Assert` function is better than a message box in two other important ways, as well. A standard message box simply has a single OK button. It lets you see that an error occurred, but there is no option to launch the debugger at the point of failure. Nor does the message box show you how the code reached the faulty state. Wouldn't it be great if you could see a stack trace with function names and line numbers and everything? And wouldn't it be nice to have the option to ignore the problem or launch a debugger? .NET's `Assert` function fits the bill nicely (see Figure 4-1).

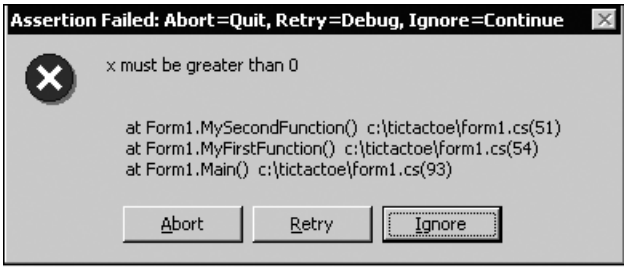

Figure 4-1. A failed assert

So when an assert pops up, investigating the problem is easy. The assert gives us a stack trace listing the exact line of the exact file, and we just look in that exact spot to see exactly what went wrong. This is nothing we couldn't have discovered by other means, but the assert presents the information automatically so we don't have to waste time gathering the information by hand. Even better, with a single click of a button, we can jump right into the debugger at this exact point in the code.

That's all an assert is. It's a debug-only warning to the programmer that something unexpected happened. An assert is just the programmer saying, "I think this condition will always, always, always be true. Anytime that assumption is not true, my code is behaving differently than I expected, and I want to be notified."

Asserts don't replace error handling and they don't replace logging. Instead, think of them as a form of documentation that gets verified at runtime. Rather than merely adding a comment that a certain variable should never be null, also add an assert. Your code might work fine today, but tomorrow a coworker will change something that invalidates one of your assumptions, and you'll have to patiently debug to figure out what broke. But if you assert every assumption you make, then a flurry of messages will fire as soon as the bad change is made. The asserts will clearly identify exactly what assumption has been violated, and then a fix should be easy.

Of course, it goes without saying that the reverse is true, too. Rather than merely adding an assert, you should add that comment, as well. Asserting that a variable will never be null is probably self-explanatory and doesn't need a comment; but asserting that a certain integer must always be in the range of 3 to 9 is much less obvious, and may require a comment to explain.

"But I Always Test My Code, So Why Do I Need Asserts?"

Even with thorough test cases, some subtle bugs might not be apparent unless you carefully study every single piece of output. You might not even notice the bug until after shipping. But if you assert every little assumption your code makes, then you'll have that much more warning if something goes wrong. Asserts take a conscious effort to ignore, and that's the point. Every time your program runs, that darned assert will pop up, reminding you to fix this bug, and the constant prompting encourages you to fix the bug sooner rather than later (if only to stop the annoyance of having to dismiss the assert each time).

Besides, even if the existence of a bug is obvious, the root cause of the problem might be occurring well before the symptoms manifest themselves. We don't want to spend time backtracking from the point of failure to the original cause. We want to launch the debugger on the very first warning signs.

Finding the Point of Failure

One day, my friend Amy asked me to help debug some scientific measurement software. Data from physical sensors was automatically fed into a complex embedded spreadsheet, and based on those calculations, feedback to the system was introduced. But over time, a subtle bug somewhere caused the values in the spreadsheet to become inconsistent in a barely perceptible error that magnified itself with each iteration until the entire system crashed several hours later. Debugging this was extremely difficult because the actual bug was happening long before the symptoms manifested themselves. Stepping through the literally millions of iterations required for the crash would simply take too long.

I believe you can never be too rich or have too many asserts. So I helped my friend write a simple function that examined every single value in the spreadsheet to ensure the data was consistent. Of course, running this function took a long time. Then we found the ten places in her code that modified that spreadsheet, and after each one, we added our assert:

```
Debug.Assert(SpreadSheetIsConsistent());
```

We recompiled in Debug mode, started the system again, and went home for the day. When we came back, an assert message was flashing on the screen. Now the code was frozen at the exact place where the data inconsistency first appeared. We had a stack trace and variable values and we now knew exactly which iteration was responsible for the initial inconsistencies in the data. Armed with that, fixing the bug was trivial.

What was the cost of adding those asserts? Well, in Debug mode, the performance hit was pretty drastic. The `Assert` function itself is fast, but the condition we were asserting on was slow. Validating the entire spreadsheet after each of the millions of iterations required huge amounts of processing time. But so what? Debug mode is for debugging—you can afford for the program to be slow then if that'll make your job easier.

The question is, what is the cost of asserts for the "real" version of the software that gets compiled in Release mode? And the answer is, absolutely no cost at all. Remember, debug asserts only run in Debug mode. The code inside the assert will be compiled away to nothingness in Release mode. So adding these debug-only asserts not only reduced a 3-day debugging task into a simple exercise, it didn't even cost the final product any performance at all.

This highlights an important lesson. Some developers think you should never write a debug-only assert that isn't also handled in Release mode. That is, some developers think we should have done something like this:

```
Debug.Assert(SpreadSheetIsConsistent());//Debug mode only
if (!SpreadSheetIsConsistent()) { //Runs in Release, too
    //Handle error
}
```

When performance in not an issue, that philosophy is definitely true. (See the sidebar "Asserts Are Not Error Handling.") If verifying data in Debug mode is good, then verifying data in Release mode must be good, too. But when performance is an issue, then that idea may be compromised. Such was the case here: Although we didn't care about performance in Debug mode, the `SpreadSheetIsConsistent` function was too time consuming to run a million times in Release mode. In cases where the validation might noticeably slow the product down, you have to be more aggressive with your validation checks in Debug mode than in Release mode. In every other case, though, you should have both error handling and asserts.

Asserts Are Not Error Handling

I've seen many developers declare that asserts encourage poor code by tempting programmers to skip writing error handling code. If so, that's a problem with lazy programmers, not with asserts. Some amateurs do indeed use asserts instead of error handling. These developers mistakenly believe that when the code runs without any asserts, there are no errors left, so why bother handling errors that will never occur? Programs developed in this fashion tend to crash-n-burn hard when confronted with the unexpected conditions of the real world. That's why some developers think asserts encourage sloppy code.

The developers who use asserts in this lazy way are half right. They litter their code with asserts and then test, knowing that asserts are a fantastic way to quickly identify and fix bugs. When your program runs in your development environment without asserts, you can indeed feel confident in it. But these developers don't realize that they'll never find all the bugs. That's why you still need error handling, too.

Asserts have absolutely nothing to do with error handling. They are not an alternative to error handling, nor a supplement, nor a special case. They serve an entirely different purpose.

Side Benefits of Asserting

Even after fixing the spreadsheet bug, we left the asserts in the code. They may be useful in the future if somebody introduces another, similar bug. Besides, the presence of those asserts gives us both a shot of confidence. When the Debug program runs and we don't see an assert, then we both feel more secure that the code is functioning correctly. In fact, that's a trick I've often used with great success. Ever see a bug that only manifests itself once in a while and you can't quite determine the pattern? Throw in a bunch of validation asserts all over your code and wait. Sooner or later, an assert will pop up to identify the bug at its root cause, which is way better than identifying the bug at its later symptom.

Naturally, if you use asserts heavily, you'll eventually assert on some condition incorrectly. Maybe when writing the code, you asserted a certain thing would never happen, but later on that assert fires on what you then realize is a perfectly valid case. So that particular assert is wrong and needs to be removed. That's another reason some developers don't like asserts: because you'll occasionally make a mistake and write an assert that shouldn't be there. Well, so what? As bugs go, an inappropriate assert is very benign. Aside from mildly annoying the developer, no harm is done to the product. Just remove that assert.

Besides, that incorrect assert taught you something about the code that you didn't know when you were writing it. The assert forced you to realize that the unexpected case was acceptable in this function. That's good information to have. Now you should review the code to make sure it can correctly handle this unanticipated case.

Don't Remove Asserts Just Because They're Annoying

I can't count the number of times a team member has told me, "Your code is asserting and it's really annoying. Can't you take those asserts out?" We look at the code and see the assert was exactly right: My teammate was doing such-and-such wrong. In fact, the assert message even clearly said in plain English what the problem was—that's the entire point of using asserts. So we fix the guy's code and the asserts go away. But then he says something like, "Well, OK, I see. But can't we comment out those asserts anyway so that they don't bother us again next time?"

I always take the time to politely explain that we should leave the asserts because they're helpful when tracking down bugs. But I privately think, "You invoked my code in an incorrect way; but rather than making you spend an hour debugging, my code automatically told you exactly what you were doing wrong before you even noticed there was a bug at all, and yet you want to remove that safety net?" It's the ostrich effect— developers cringe at asserts because it proves they have a bug that they can't ignore. Whereas if there were no assert, maybe nobody would notice the bug until after the code shipped. Then they wouldn't have to deal with it because that angry customer will be somebody else's problem.

Don't be like that. If someone went through the trouble to write an assert, they probably had a good reason for doing it. Respect that reason and assume any assert you see is telling you something important.

.NET's Debug and Trace Classes

Up to this point, I've been speaking of the debug-only `Assert` method found in .NET's `System.Diagnostics.Debug` class. Now would be a good time to mention the `Debug` class's twin sister, `System.Diagnostics.Trace`. Both provide excellent logging facilities (described in Chapter 5) as well as the `Assert` method. In fact, both classes have the exact same set of methods. The one difference is that methods from the `Debug` class only work when the code is compiled in Debug mode, but the methods from `Trace` work in either Debug or Release mode. So when I talk of debug-only asserts, I'm referring to the `Debug` class. If you used the `Trace` class, then your asserts would appear all the time, not just in Debug mode.

Why would you ever use the `Trace` class's `Assert` method? What value is an assert in Release mode? Good question. Personally, I've never found much use for the `Trace` version of `Assert`. An assert is meaningful to the developer; it is not usually meaningful to an end user. Users can appreciate error messages like "The file you selected is invalid. Please select a different file", because even though that error message doesn't describe the problem very well, it at least identifies what to do. On the other hand, the assert in Figure 4-2 would be excellent for a developer, but do you think your average user would understand what it means?

Figure 4-2. What not to show your users

Also, the dialog box displayed by the `Assert` function gives the options to debug, ignore the problem, or abort. You probably don't want to present those options to your users. Does your average user have the necessary skills, tools, and source code needed to debug your program? Does she have the necessary information to decide whether the assert can safely be ignored or whether she should abort? Probably not.

If you encounter a severe error in Release mode that cannot be hidden by the error handling code, you are best served by displaying a regular message box with a human readable description of the problem and a simple OK button. Maybe that error message is merely, "A fatal exception occurred. Please contact tech support." That's still preferable to showing the user an assert. At least this error message tells the user what to do. All in all, I'd recommend avoiding the `Trace` class's `Assert` method. The `Trace` class is primarily used for logging, not asserting.

In any case, regardless of whether you use the `Debug` or `Trace` class, the `Assert` function comes in three flavors:

```
Assert(bool condition)
Assert(bool condition, string description)
Assert(bool condition, string description, string extendedDescription)
```

They all behave similarly—they check whether the condition is true and display a message if it isn't. The difference is what they display. All three versions display a stack trace leading up the assert, and that's all you see with the first overload. The second overload shows a stack trace, too; but it also displays a user-defined string to describe the assert. For example, maybe you want the assert to show you text like `"Error: input parameter was null"` so that you can immediately identify the problem without having to launch the debugger. Finally, the third version is useful if you want the assert to show both a short description (`"The parameter was null"`) as well as a longer description (`"This error usually means that the calling function failed to do thus-and-so. To fix it, do blah blah blah."`) I normally use the second version, but the choice boils down to how much you want to type.

Asserts Are Compiled to Nothingness in Release Mode

There's one mistake every developer makes once. Consider the following code snippet:

```
bool didItWork = DoStuffAndReturnTrueIfItWorked();
Debug.Assert(didItWork);
```

Anyone who is new to asserts will be tempted to save a line of code by rewriting that as

```
Debug.Assert(DoStuffAndReturnTrueIfItWorked());
```

Test that code in Debug mode and everything works fine. But it will produce a tremendous error in Release mode, where debug asserts are compiled away into nothingness to maximize performance. So even though this code works great in Debug mode, when running in Release mode, the `DoStuffAndReturnTrueIfItWorked` function will never be executed because the assert will make it disappear.

Make sure that you don't put any function that does important stuff by side effect in an assert. Put that function outside the assert, and then call the `Assert` function on the return value.

Using Asserts Aggressively

Some developers believe in asserting only fatal errors—the kind of errors that would normally cause a program to crash. For instance:

```
using System.Diagnostics;
using System;

class AssertTest {
    public static object FunctionThatReturnsAnObject() {  … }
    public static void Main() {
        object o = FunctionThatReturnsAnObject();
        //Program will surely crash if that ever returns null.
        Debug.Assert(o != null);
        if (o == null) {
           throw new Exception("Way bad");
        }
        …
    }
}
```

The preceding assert is correct usage. Not only does the assert warn about problems at the point of failure, but there's also error handling to deal with the case in Release mode where asserts don't fire. Nothing at all wrong with this code. But some developers think this is the only type of code where asserts should be used. I disagree. I believe you should also assert on non-fatal errors, too. In fact, you should use asserts on just about everything. Use the power of asserts to track down every bug, not just the ones that crash your program. For example, suppose you're writing code for an online store. Your code fetches the list of products your store sells from a database using ADO.NET:

```
using System.Data;
using System.Data.SqlClient;
using System.Diagnostics;
class SqlTest {
    public static void Main() {
        string dbConnStr = "Server=(local);Database=myDb;uid=myUid;pwd=myPwd;";
        SqlConnection dbConn = new SqlConnection(dbConnStr);
        dbConn.Open();
        string selectStr = "SELECT productName FROM productsTable";
        SqlDataAdapter da = new SqlDataAdapter(selectStr, dbConn);
        DataSet ds = new DataSet();
        da.Fill(ds, "productsTable");
```

```
        //We expect that table will NEVER be empty, so assert:
        Debug.Assert(ds.Tables["productsTable"].Rows.Count > 0);
        foreach (DataRow row in ds.Tables["productsTable"].Rows) {
           … //Do stuff with each row
        }
        dbConn.Close();
    }
}
```

Suppose the list of rows in `productsTable` came back empty. That's not necessarily a bug in this function, since it's perfectly legal for a database table to be empty. And it's certainly not a fatal error—your GUI won't crash; it will simply display zero products. But is it reasonable to think your store will ever sell zero products? Probably not. If that `DataSet` ever contains zero rows, then something is clearly wrong. Someone accidentally deleted the contents of your database, or maybe your database query was wrong. Either way, it's definitely an error you want to know about. Add an assert that the number of rows will be greater than zero. Add asserts on everything you think will always be true, no matter how minor or non-fatal it is. Even if you think, "Oh, this could never happen," add an assert anyway.

Or what about this one? Suppose your program saves some data to an XML configuration file and later reads it in. Since you wrote that configuration file, you know there ought to be a tag indicating whether the user wants the AdvancedView feature turned on or not. Suppose your program reads in that file and realizes that tag is missing. Is this a fatal error? Absolutely not—well-written code should be able to easily deal with this case by just assigning the default behavior to the AdvancedView feature. But is this an unexplained bug that should worry the programmer? Definitely. If the file is missing this line, then who knows what else is wrong, too? Even though this is a non-fatal error, it's still something that needs an assert.

Asserting Performance Bottlenecks

Asserts can also warn you when code performance falls below a certain threshold. Suppose you have some speed-critical code in an inner loop. If the timing of that section ever exceeds X milliseconds, then you might add an assert to notify you that your code is running slower than expected. That'll give you a head start on deciding where to optimize. The point here isn't to replace a traditional performance profiler. The point is just to make sure that the developer will be instantly informed if this unexpectedly bad performance ever happens.

```
using System.Diagnostics;
class TimingTest {
    public static void Main() {
        #if DEBUG
            const int maxExpectedTime = 500; //.5 seconds
            System.DateTime startTime = System.DateTime.Now;
            #endif
        //First speed-critical section of code here
            ...
        #if DEBUG
            System.DateTime endTime = System.DateTime.Now;
            System.TimeSpan elapsedTime = endTime.Subtract(startTime);
            Debug.Assert(elapsedTime.Milliseconds <= maxExpectedTime,
                "First section took too long");
            startTime = System.DateTime.Now;
            #endif
        //Second speed-critical section of code here
            ...
        #if DEBUG
            endTime = System.DateTime.Now;
            Debug.Assert(elapsedTime.Milliseconds <= maxExpectedTime,
                "second section took too long");
        #endif
    }
}
```

In the preceding code snippet, there are two pieces of speed-critical code. A real-world example might have four or five speed-critical "hot spots." When you run this program, you notice the feature is slow and you suspect the bottleneck is occurring in one of the known speed-critical sections—but without running a performance profiler, you couldn't tell which one. Yet these asserts will tell you if any section takes longer than expected, so that helps pinpoint which section is taking too long. And what if none of the asserts fire? Then that would prove the bottleneck is not in any known hot spots, so get out that profiler to identify the real culprit.

You probably don't have a firm estimate for the number of milliseconds a piece of code should require—developers tend to be very bad at predicting such things. But you can make some good, conservative estimates. Just ask how long a typical user would be willing to wait for a feature to run. Two seconds? Thirty seconds? Put an assert around the feature stating it should never take longer than your chosen value. Don't forget, though, that various factors make it difficult to reliably time intervals less than about .2 seconds. Better to assert on longer intervals.

Asserting to Prepare for the Future

I once inherited the code of a management tool for Microsoft Exchange. The authors intended to support both Exchange 5.5 and Exchange 2000, but time pressures forced them to defer support of Exchange 2000. APIs for the two versions were similar, but there were a few substantial differences. So when I was assigned to take over this code base and add support for Exchange 2000, I feared I'd have to spend weeks searching to find every single bit of Exchange 5.5–specific code.

Imagine my pleasant surprise when I discovered the previous authors had made heavy use of asserts to identify those sections.

```
Debug.Assert(version == Exch55, "This won't handle Exchange 2K yet");
```

So I did a first pass through the code to add support for Exchange 2000. Naturally, I missed a few places that needed to be changed. But when I ran my first test against an Exchange 2000 server, asserts popped up on all the places I missed. I didn't have to spend hours debugging to figure out all the problems—all I had to do was run the code and then the asserts instantly lead me there. It's times like that that make you want to send a dozen roses to the people who maintained the code before you.

Prepare for the future. Anytime you have code that works fine now but will require changes if somebody does such-and-such in the future, then add asserts to fire when such-and-such happens. That way the person who implements such-and-such will be guaranteed to notice he needs to change your code, too. He'll thank you for it.

Don't Assert Legal Cases

But it's important to make sure every assert you write is truly a case that should never happen. Some developers write asserts for unusual but legitimate cases that do occasionally happen in a "normal" run of the program. That's a bad habit to get into, though. For example, suppose your online store presents a textbox where customers can type the name of a product, and then you have a function that looks up the price of that product:

```
using System.Collections;
using System.Diagnostics;
class PriceLookupClass {
    Hashtable pricesTable = …;
    string GetPrice(string productName) {
        string price = (string)pricesTable[productName];
        if (price == null)
            Debug.Assert(false, "Error: Product not found");
        return price;
    }
}
```

The problem here is that it's perfectly legal for a customer to type the name of a product your store doesn't carry. It's not a bug you need to fix. You definitely need to handle that case by displaying some kind of "Sorry, we don't carry that product" message, but a failed lookup in the hashtable is not a bug. Remove that assert immediately. Anytime you see an assert pop up, you ought to think, "Uh oh, something's wrong and I need to figure out what!" But if you allow yourself to start thinking, "Oh, that assert doesn't really mean anything," then you'll never have the motivation to fix the problems indicated by asserts again.

On the other hand, suppose you didn't offer the user a textbox and instead forced her to choose from a drop-down list of products that you know your store carries. Would an assert be appropriate then? Yes! With a drop-down list, it would be impossible for the user to choose a product you don't carry—so if the hashtable lookup failed then, it would mean you really did have a bug. If that's what your GUI looks like, then leave the assert in this function. Notice the implication of that statement. You can't base your asserts just on the surrounding code; you have to use "big picture" logic to understand which circumstances are possible and which aren't.

Anytime you see an assert pop up, you need to either fix it or remove it—leaving it untouched is a sin. But make sure you think twice before you remove the assert. Removing it is sometimes the right thing to do, but first make certain you understand the original purpose of the assert and why it is no longer valid. Don't give in to the temptation to remove an assert just because you don't understand

why it's failing and things seem to work if you ignore it. The person who wrote the code had a reason for believing that condition would always be true, so make sure you understand it before removing the assert.

When Asserts Can't Be Easily Used

Marvelous as asserts are, there are cases where it's difficult or impossible to use them. The basic issue is that asserts halt the execution of the program—that way you have plenty of time to analyze the assert and decide whether or not to deal with it. But in some situations the standard assert can never be acknowledged. Sometimes these situations can be worked around, and sometimes not.

Fortunately, though, the .NET architecture is flexible enough to cope with almost any situation. Can't use a standard assert? When push comes to shove, you can write your own. First, we'll discuss the situations where standard asserts are problematic. Then, we'll write a customized assert that will handle these cases the way we want.

The standard asserts can't be used easily in the following types of programs:

- Windows services
- ASP.NET
- Remote objects

Asserts with Windows Services

The first place where standard asserts can't be easily used is in Windows services.

NOTE Don't confuse Windows services with Web Services—they are entirely different things. Web Services are new to .NET and allow distributed components to communicate through XML. Windows services have been around since Windows NT and are a way of making a special "background" process that has a few differences from a normal process.

Windows services normally don't have user interfaces. There are exceptions, but in general, services sit unnoticed in the background. There is a good reason for this. Many services such as Microsoft Exchange or SQL Server run on heavy-duty

computers locked in a company's server lab. Generally, no one looks at those machines on an average day, so if those services ever displayed a message box and waited for acknowledgment by the user, then it might be a long while before anyone noticed.

So when you write a service to display a "Hello World" message box, that message box will open up on a hidden window station that no one will ever see. The program will be blocked forever, waiting on someone to acknowledge that hidden message box. A similar thing happens when a service displays an assert. Your service would freeze up if it tried to display the assert, so .NET services will normally avoid the problem by ignoring the assert and not displaying anything. It's as if you never wrote the assert at all.

TIP Actually, the preceding paragraph isn't entirely accurate. The `MessageBox` class allows you to specify some flags called `MessageBoxOptions`, and one of those flags is called `ServiceNotification`. If you go through the extra trouble of setting this flag, your services can display message boxes. We'll use this trick later on to write our own custom assert.

Asserts with ASP.NET Pages and Web Services

The problems with asserts in Windows services also applies to ASP.NET web pages and Web Services. But that should come as no surprise, especially when you realize the ASP.NET engine is a service itself. If a web page fired an assert, then who would see it? It may be that no one is logged on to the web server machine, so no one would be able to acknowledge the assert and then the process would be hung. To avoid this problem, ASP.NET pages and Web Services normally ignore asserts.

There's nothing we can do to fix this in the standard assert. Luckily, though, we can work around the problem by writing our own customized assert. How would we want this to work? Well, if the web page is being accessed by a user on a different machine, then we probably don't want to display any assert, since that user probably doesn't have access to the web server. On the other hand, we would want to display an assert if the web page request came from the local machine, because that user is obviously in a position where the assert could do some good. We'll discuss writing this customized assert shortly.

Multi-Tiered Services

There's another option that allows the use of asserts in services. This option also results in cleaner code, and simplifies debugging service startup code, a notoriously difficult task. Consider: It's good software practice to separate code into tiers, and it's not very hard to separate the "service" infrastructure code from the "stuff you care about" logic code. If you did that, you could write two versions of your program that use the same logic code—one is a service and the other is an application.

What does that gain? There are no restrictions on asserts in an application. When testing the code during development, run the application version to see all the asserts. Once your code has stabilized, switch to running the service version that will eventually be shipped. This gives you the best of both worlds—run the same code as either a service or an application, whichever is more convenient.

Accomplishing this is easy. Your Windows Service exposes only a few methods: `OnStart`, `OnStop`, `OnPause`, etc. Rather than writing your logic code inside these methods of the `ServiceBase` class, define a standalone helper class that does all the work. Then your `ServiceBase` class can merely delegate to the helper class. Or, you could write a standard app to do the same thing:

```
//Service version
public class MyServiceClass : System.ServiceProcess.ServiceBase {
    ...
    private MyServiceLogic logic = new MyServiceLogic();
    protected override void OnStart(string[] args) {
        logic.OnStart(args);
    }
}
```

```
//Application version
public class MyApplicationClass : System.Windows.Forms.Form {
    …
    private MyServiceLogic logic = new MyServiceLogic();
    private void startButton_Click(object sender, System.EventArgs e) {
        string[] args = …;
        logic.OnStart(args);
    }
}

//The class that actually does the work
public class MyServiceLogic {
    public void OnStart(string[] args) {
        …
    }
    …
}
```

As an added bonus, this approach makes it easier to debug the service startup code, too. Traditional services can only be started via the Service Control Manager or the `net start` command, and both of those prohibit starting code through the debugger. You can always attach the debugger to an already running service—but suppose you're debugging a problem with the service startup itself? Visual Studio .NET doesn't support that very well. But architecting your service in this tiered approach can greatly simply that task.

Asserts with Remote Objects

Objects running on remote machines also present a problem for asserts. Normally, an assert pops up on the screen of the computer where that code is running. But if you invoke code on a remote server and if that code asserts, then the remote code will be stuck waiting on someone to acknowledge the assert. Meanwhile, the code running on your local machine will be stuck waiting on a response from the remote machine, and that response may never come. If the assert were displayed on your local machine instead of the remote machine, then you could acknowledge it; or if you had a coworker monitoring the remote machine, then he could acknowledge it. But without that, the system will be hung.

During development, the "remote" machine will often be in the same building as the local machine. If so, then this problem may be tolerable. Anytime your program takes an inordinate amount of time returning from a function call to the remote object, check the remote machine to see if it's caught in an assert. However, if you don't have easy access to the remote machine, then you will likely have problems including asserts in the remote object code. Watch out for this problem when using remote objects.

Customizing Asserts with a TraceListener

The `Debug` and `Trace` classes can easily be customized to behave any way you want. Rather than actually doing the work of the assert themselves, these classes merely delegate to a helper class called `System.Diagnostics.TraceListener`. We'll discuss `TraceListeners` further in Chapter 5, but for now, you just need to know that by writing your own `TraceListener` class, you can customize how your asserts behave. For instance, if you wanted your asserts to e-mail you rather than displaying an error message, you could easily do that.

Let's write a `TraceListener` class to customize our asserts for ASP.NET. (Don't worry if you don't yet fully understand how to use `TraceListener`—we'll come back to the topic in the next chapter.) Remember, our goal for ASP.NET asserts is to display them when the user is browsing the page from the web server machine, but not when the user is browsing from a different machine. To detect that, we just need to compare the IP address of the web server with the IP address of the web request.

CAUTION Technically, detecting the IP address can be slightly complicated—some advanced machines may have multiple NICs that create multiple IP addresses. But we'll ignore that here since asserts are only needed on development machines anyway. If your development machine has multiple NICs, then you can take that into account during your testing.

After checking the IP address of the web request, we need to display an assert dialog box in a way that the service-based nature of ASP.NET won't reject. It turns out this can be accomplished by using the `MessageBoxOptions.ServiceNotification` flag as demonstrated here:

```
using System;
using System.Web;
using System.Diagnostics;
using System.Windows.Forms;
public class AssertClass : DefaultTraceListener {
    public override void Fail(string message) {
        Fail(message, null);
    }
    public override void Fail(string message1, string message2) {
        HttpRequest req = HttpContext.Current.Request;
        //If this object is being used in a non-web based application,
        // then we can safely display an assert.  Otherwise, only
        // show assert if user is logged in to local machine.
        if (req == null ||
            req.UserHostName == req.ServerVariables.Get("LOCAL_ADDR")) {
            //Display the stack trace of the failed assert
            StackTrace t = new StackTrace();
            string s = t.ToString();
            string displayMe = message1 + "\n" + message2 + "\n" +
                "Launch debugger?\n\n" + t.ToString();
            DialogResult r =
                MessageBox.Show(displayMe, "Assert failed!",
                    MessageBoxButtons.YesNo,
                    MessageBoxIcon.Error,
                    MessageBoxDefaultButton.Button1,
                    MessageBoxOptions.DefaultDesktopOnly);
```

```
            //If the user pushed yes, launch the debugger
            if (r == DialogResult.Yes)
                Debugger.Break();
        }
    }
}
```

Wasn't that easy? The `Debug` and `Trace` classes are designed to delegate calls of the `Assert` method to the `TraceListener`'s `Fail` method. There aren't any tricks to the preceding code—we just check if the user is on the local machine, and if so, then we display a message box using the `MessageBoxOptions.DefaultDesktopOnly` flag. Finally, if the user asks, then we'll launch the debugger. A very few simple lines and now we can use asserts in our ASP.NET pages and Web Services. All that remains is to tell ASP.NET to use our new `TraceListener` class instead of the default. We can do this in our ASP.NET application's startup method:

```
using System.Diagnostics;
public class Global : System.Web.HttpApplication {
    …
    protected void Application_Start(Object sender, EventArgs e) {
        Debug.Listeners.Clear();
        Debug.Listeners.Add(new AssertClass());
    }
}
```

When the Customized Assert Fires

Our customized assert looks very much like the standard assert (see Figure 4-3).

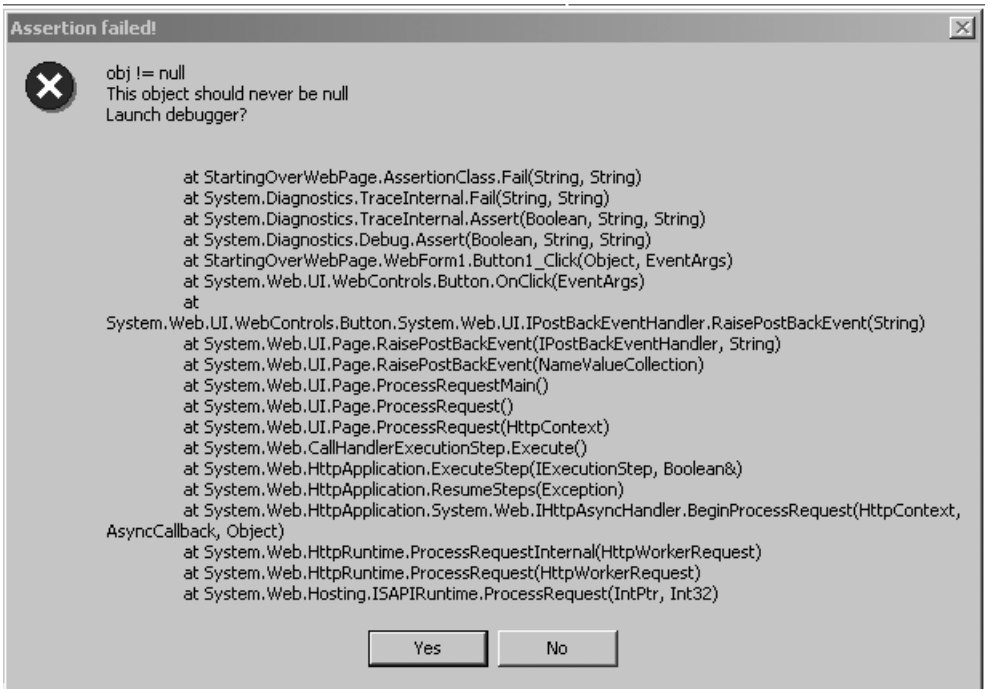

Figure 4-3. The customized assert

If you accept the option to launch the debugger from that assert, then Visual Studio .NET will ask you what type of debugger you wish to use. Accept all the defaults, and you'll be dropped into the VS .NET debugger. There's only one thing to be aware of. As shown in Figure 4-4, the debugger will be started at your assert code's call to `Debug.Break` (which you don't have debugging symbols for anyway), whereas you want to start debugging at the function that invoked the assert. That's easily solved. Just press the "Step out of" debugging button a couple times until the call stack jumps up to the place you want to be.

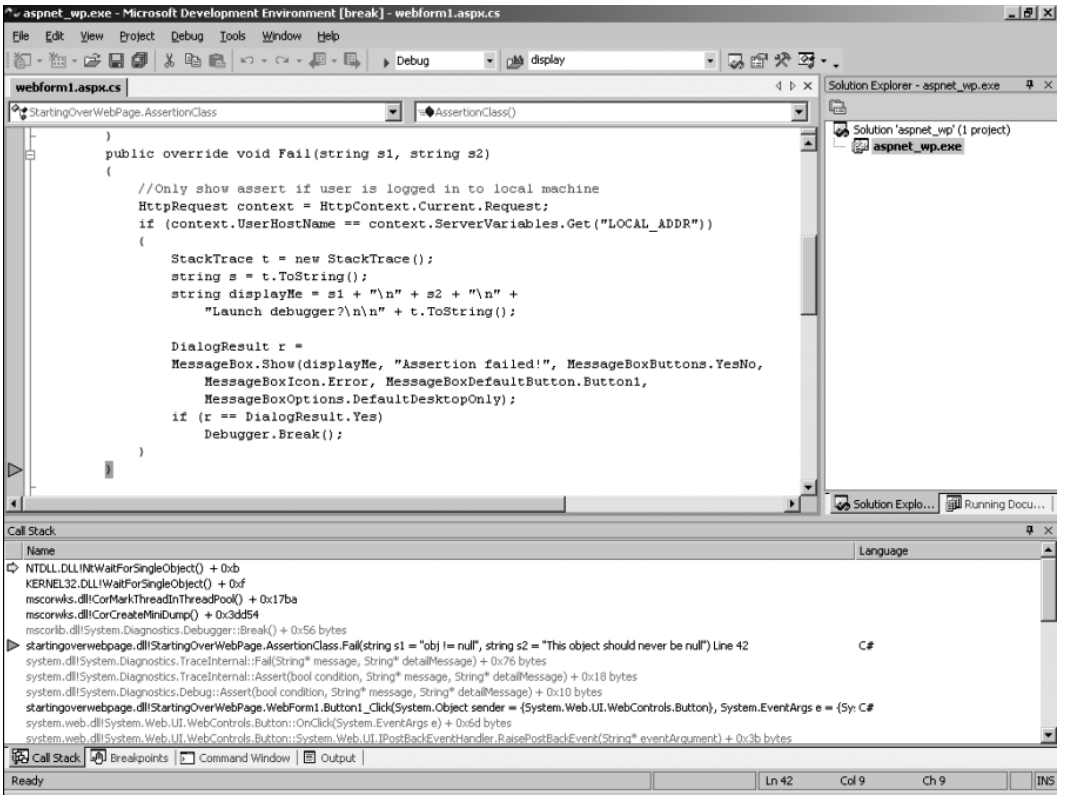

Figure 4-4. The call stack after invoking our customized assert

Using Customized Asserts in a Service

Modifying this assert to work with services is easier or harder, depending on how your service will be used. We were able to detect that the user and ASP.NET page were on the same machine by using the `HttpRequest` class. However, that trick won't work for service requests. If your service will only be accessed by users logged on to the local machine, then you can assume the assert should always be raised in Debug mode, and simply shorten the `TraceListener` code to always display the assert message box. But if users might be connecting to this service from other machines, then you need to be careful or else those users might get confused when an assert on the service machine causes their client process to hang, forever waiting on a response.

But even if you find using asserts difficult in a few situations, don't let that blind you to the thousands of other situations where asserts can be one of your most powerful weapons. Few other techniques can pinpoint bugs as effectively as debug asserts, and learning to use asserts correctly is a skill every programmer should develop.

Highlights

- Asserts notify you the instant one of your assumptions about the code is invalidated. Think of asserts as a form of documentation that gets verified at runtime. Running your code and not seeing any asserts gives you a shot of confidence that the code is probably behaving correctly.

- Asserts can find bugs at the source of the problem, not just at the later symptom. The root cause of many bugs occurs long before problems are apparent in the program's output. But well-written asserts can launch a debugger at the exact moment when errors begin to happen, saving you hours of debugging time.

- Asserts are not for error handling. Asserts help identify bugs so you can fix them, but you still need to write error handling code for those errors that inevitably slip through. In most situations, each assert should be immediately followed by code to deal with the problem identified by that assert, just in case it ever comes up in Release mode.

- Asserts normally run only in Debug mode. Remember any code inside an assert will be compiled away to nothingness in Release mode. You can have asserts run in Release mode by using .NET's `Trace` class, but there's not much point in doing so.

- Make sure each assert is a legitimate error. If you're asserting on conditions that are rare but legal, then you're doing something wrong. As assert should represent an honest-to-goodness error that needs to be investigated.

- Be careful using asserts in ASP.NET, distributed objects, or Windows services. Standard asserts will not work in these situations because they could potentially result in a process being blocked, since there may not be anyone on the remote machine available to acknowledge the assert.

- You can write your own `TraceListener` class to customize asserts in whatever way you like. This is one of the best ways to use asserts in ASP.NET or Windows services.