.NET Gotchas

Venkat Subramaniam
venkats@agiledeveloper.com
http://www.agiledeveloper.com/download.aspx

Modified 08/28/2005: When I wrote this short article, I did not realize that I would publish a book with the same title some day! I got inspiration to write a book on this topic seven months later. The book .NET Gotchas, published by O'Reilly in May 2005, has 75 gotchas and is about 400 pages long☺

Modified 08/28/2005: Brian Grunkemeyer of the Microsoft BCL team pointed out error in this document for the Dispose pattern. Please note the change on page 5.

(I have not changed anything in this document. Where a change is needed, I have placed a comment on the side.)

Abstract

Those of us programming on the .NET framework using one of the .NET languages and Visual Studio have come to realize the power and increased productivity that comes with it. Like any development, however, there are things that one should pay attention to. This article presents a number of things that we, as developers, would benefit from keeping in mind. This article presents 40 items we need to be aware of in the area of CLR/Framework, Visual Studio, Compiler, Language, Garbage Collection, Inheritance, Multithreading, COM-Interop, ASP.NET and Web Services.

CLR/Framework Gotchas

1. *Watch out for language specific aliases*
   Common Type System (CTS) defines the types that all .NET languages use. Examples of these are System.Int32, System.Single, etc. However, each .NET language defines aliases to these types. For instance, int in C# and Integer in VB.NET are aliases to the System.Int32. These aliases correspond to the types C++ developers (in the case of C#) and VB developers (in the case of VB.NET) are familiar with. However, one should keep in mind that the size of the data types is not consistent with what one might be familiar with. For instance, Integer in VB is not the same size as Integer in VB.NET and long in C++ is not the same size as long in C#.

2. *Watch out Value-Type vs. Reference-Type when assigning*
   The effect of code r1 = r2; depends on whether r1 and r2 refer to value types or reference types. If they are value types, the value of the object r2 is copied to the object r1. However, if they are references, r1 refers to the same instance that r2 refers to. This may lead to confusion as one might not know which type it is by looking at the above assignment.

3. *Avoid using + to append Strings*
   If your intent is to append strings, you may improve performance by using a StringBuilder instead of String[1].

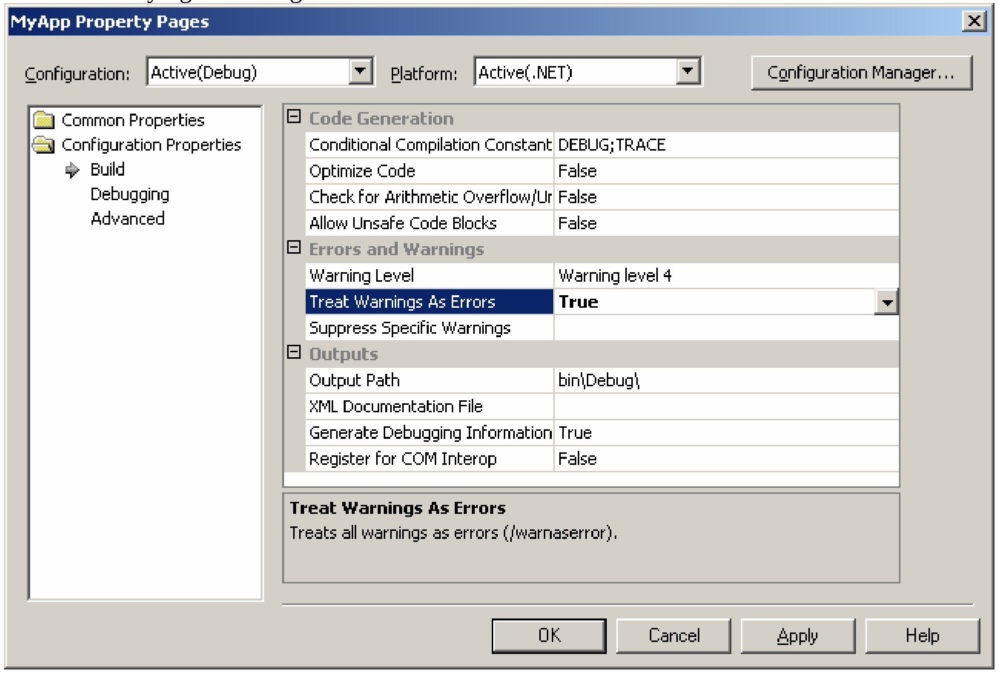
4. *Watch out for delegate being null*
   If you are implementing a component that generates events, use caution when firing an event using a delegate. If no delegates have been registered an exception is thrown when

you generate an event. Check to make sure the delegate is not null (Nothing in VB.NET) before firing the event.

Visual Studio Gotchas

5. *Treat Warnings as Errors*

   When you compile your code in VS.NET, if there are no errors, you will notice a message similar to "Build: 1 succeeded, 0 failed, 0 skipped" appear in the Output Build window. However, if there were warnings, these get hidden in the Output Build window. Some of these warnings are severe and should have been actually treated as errors. For instance assume a method is virtual in the base class. If one writes a method with the same name in the derived class, but forgets to mark it with the keyword override (overrides in VB.NET), then a warning is generated as "warning CS0114: 'MyApp.Derived.fn()' hides inherited member 'MyApp.Base.fn()'. To make the current member override that implementation, add the override keyword. Otherwise add the new keyword." Unfortunately, the compiler treats the function fn of Derived as *new* instead of treating it as *override*. A developer keen on writing robust and dependable code would not want to treat warnings lightly. It is highly recommended that you set "Treat Warnings as Error" on your project by right clicking on the project within solutions explorer and modifying the setting as shown below:



Code Generation	
Conditional Compilation Constant	DEBUG;TRACE
Optimize Code	False
Check for Arithmetic Overflow/Ur	False
Allow Unsafe Code Blocks	False
Errors and Warnings	
Warning Level	Warning level 4
Treat Warnings As Errors	True
Suppress Specific Warnings	
Outputs	
Output Path	bin\Debug\
XML Documentation File	
Generate Debugging Information	True
Register for COM Interop	False

Treat Warnings As Errors
Treats all warnings as errors (/warnaserror).

Compiler Gotchas

6. *May want to set references to Nothing after last use*

   The C# compiler sets a reference to null after the last usage. However, the VB.NET compiler does not appear to do so. While this may not be an issue most of the time, it may be of concern if a method creates an object and goes into an extended computation cycle after that. Assume we have a code where an object is created, used and then the code goes into a computation cycle that may take significant amount of time. Assume that the reference is not used any more after its initial use. If the garbage collector were to run while the function is in the middle of the computation, in C#, the object may be garbage collected since the compiler has implicitly set the reference to null. However, we have noticed that this does not happen in VB.NET code. One may want to set the reference to Nothing after the last use in order to facilitate speedy cleanup in cases where this may be significant.

Language Gotchas

7. *Use aliases sparingly*
   You may define your own aliases for types as in the following C# syntax:
   using Number = System.Int32;
   While this may be convenient at times, excessive use of this would make it hard for others to read and understand the code.

8. *Do not create properties with same name that differ only in case in C#*
   Of course this should generally be considered a bad practice, period. One reason especially this should be avoided is that VB.NET is not case sensitive. If you were to write two properties in a C# class, say myproperty and MyProperty, then when referenced from VB.NET, only the first property is visible. It hides any other property with the same name in its scope.

9. *Do not write a public copy constructor and do not rely on the MemberWiseClone*
   If you have a need to copy your objects, write a protected copy constructor and invoke it from your own implementation of clone[2].

10. *Do not invoke static (shared) members using an object reference*
    In C# a static member may only be accessed on the class. It does not allow one to access these using an object reference. However, in VB.NET, this restriction is not in place. Given that static/shared members are not polymorphic, it is easy for one to be confused if static/shared methods are invoked using an object reference, especially if the static/shared methods are part of the derived class as well. In VB.NET, it is better that you do not use an object reference to access shared methods/members. Instead, use the class to access these.

11. *Avoid Operator Overloading*
    C# allows you to overload operators. However, language like VB.NET does not support operator overloading. For your code to be CLI compliant, you are required to provide a regular method for each overloaded operator. An example of this is the += operator on Delegates and the related Combine method. Overloading operator is supposed to make the code more intuitive and easier to understand. On the contrary, there are several issues related to overloading. For instance, overloading true requires overloading false in C#. Also, the && operator is implemented through a combination of overloaded true, false and &. Arbitrarily overloading operators largely increases complexity to the extent than its worth is questionable.

Garbage Collection Gotchas

12. *Do not rely on Finalize being invoked, instead Dispose objects yourself*

One can not predict when exactly Finalize will be called on an object. Any resources not release will be held until Finalize is called and this may lead to some undesirable effects. It is better for one to invoke the Dispose method on an object (assuming IDisposable is implemented) instead of relying on the Finalize. In C#, utilize the *using* clause to ensure objects are disposed properly.

13. *Invoke the base's Finalize from within your Finalize method in VB.NET*
    In VB.NET, from within your Finalize method invoke the base class' Finalize method. If you are using C#, however, you do not invoke the base class' Finalize method.

14. *Implement Dispose on an object and suppress finalize in it*
Call to Finalize involves overhead. If resources are cleaned up properly, there is no reason for Finalize to be executed on an object. In your Dispose method, invoke GC.SuppressFinalize() method to tell the CLR not to bother finalizing the object.

15. *Within Finalize method, do not access any managed resources*
Let's say an object A has a reference to an object B and there is a reference on the stack to the object A. Now, both objects are considered to be reachable. If the reference on the stack goes out of scope or it is set to null (Nothing in VB.NET), then the two objects are considered to be unreachable. When unreachable objects are garbage collected there is no guarantee on the order in which they will be Finalized. Accessing other managed resources/objects within Finalize may lead to unpredictable results.

16. *Follow the Design Pattern for Dispose*
If a user of an object calls the Dispose method both managed resources and unmanaged resources needs to be Disposed. However, if a user forgets to call Dispose, when the Finalize is eventually invoked on the object, only unmanaged resources must be released (see item #15). Understanding the Dispose Pattern[3] helps us with proper implementation of the Dispose method.

> Modified 08/28/2005: Brian Grunkemeyer of the Microsoft BCL team wrote a couple of days ago: "In this document: http://www.agiledeveloper.com/articles/DotNetGotchas.pdf You have an improper example of the Dispose pattern. Dispose(bool) should be protected, not public. Dispose(void) should call GC.SuppressFinalize(this) after calling Dispose(bool). You can check on the web for updated design guidelines – Joe Duffy, myself, and a few others cleaned these up about 6 months ago."
>
> Thanks Brian. Folks, please refer to http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconFinalizeDispose.asp for the correct example. [This error is only in this online example and not the .NET Gotchas book☺]

Dispose for the Base class:

```
public class Base : IDisposable
{
        //...
        public virtual void Dispose(bool disposing)
        {
                if (disposing)
                {
                        // cleanup/ dispose managed resources here.
                }

                //cleanup unmanaged resources here
        }

        public void Dispose()
        {
                Dispose(true);
        }
```

Comment [VS1]: See note above.

Comment [VS2]: See note above.

```
        ~Base() // Replace with Finalize method in VB.NET
        {
                Dispose(false);
                // In VB.NET, invoke MyBase.Finalize();
        }
}
```

Dispose for the Derived class:

```
public class Derived : Base
{
        public override void Dispose(bool disposing)
        {
                if (disposing)
                {
                        // cleanup/ dispose managed resources here.
                }

                //cleanup unmanaged resources here

                base.Dispose (disposing);
        }
}
```

Inheritance Gotchas

17. *Use the "is" operator sparingly*

    The "*is*" operator (TypeOf in VB.NET) allows us to query if a reference is referring to an object derives from or implements a certain type. While this may seem very convenient, in general, this may lead to code that is not extensible. We may end up violating Open-Closed Principle[4] (OCP). Use it sparingly and only in cases where you will not be violating OCP.

18. *Do not override a method or declare it new if base method is not virtual*

    If a method declared in the base class is not virtual, then the method is not overridable. Declaring a derived class method with the same name (and marking it as new) will result in hiding. See item #19 and #20.

19. *Do not hide methods when subclassing*

    C# allows you to hide a method when subclassing. Assume that a method is declared virtual in the base class. If a method is overridden (with the override keyword) in the derived class, then on an object of derived class, the derived class method is invoked if the type of the reference is derived type as well as base type. However, if in the derived class you write the method but declare it as *new*, the method ends up hiding the base class method. Invoking the method using a derived class reference results in a call to the derived class method; however, invoking the method using a base class reference results in invocation of the base class method. The resulting behavior of the code may be undesirable.

20. *Remember to mark method as override*

If a method is declared in the base class as virtual and you implement a method with the same name and signature in the derived class, remember to mark the derived class method as override. If you forget to mark it as override, the compiler gives a warning and treats the method as *new*. Refer to item #19.

21. *Do not mark a derived class method as virtual if the base class method is virtual*
    Programmers coming from C++ are used to mark methods as virtual in the derived class when overriding virtual methods from the base class. However, in .NET, if the method is marked as virtual in the derived class the compiler treats it as a *new* method instead of *overriding* it. This results in the hiding of the base method instead of overriding. Mark the derived class method as override.

Multithreading Gotchas

22. *Do not use certain methods of Thread class*
    Certain methods of Thread class are non-deterministic. By the time you process the response from these methods, the thread's state may have changed. For instance calling ThreadState or IsAlive may indicate that a thread is alive, however, by the time you process the response the thread may have quit.

23. *Do not call Start on a Thread more than once*
    Calling Start on a Thread that has been started, terminated or aborted results in a ThreadStateException being thrown. If you need to start executing a method again in a new thread, then create a new Thread object and call start on it.

24. *Make the thread a background thread*
    Your application keeps running while there is at least one non-background thread running. By default, threads in .NET are created as non-background threads. This may not be desirable. When creating a thread and before starting it, ask yourself if this should be a background thread? If so, set the IsBackground property to true before you invoke the start method.

25. *Do not use Suspend and Resume on Thread*
    You have no idea what the state of the thread is and which method it is executing when you call Suspend on a thread. What if the thread is holding some locks while you call Suspend? Using these methods is the easiest way to create deadlocks in your application.

26. *Use caution when calling Interrupt on a thread*
    When you call interrupt on a thread, if the thread is blocked in a Sleep, Join or Wait, a ThreadInterruptedException exception is thrown. However, if the thread is not currently blocked, the exception is not thrown on the thread and it is not interrupted until it gets block in a Sleep, Join or Wait. Calling Interrupt may not result in the thread being interrupted for a long time or never.

27. *Know that ThreadAbortException is a special exception*

When a Thread is Aborted, ThreadAbortException is thrown on the thread. While the thread may have a catch block to handle ThreadAbortException (and can take care of proper cleanup and termination in this block), the ThreadAbortException is automatically raised again at the end of the catch block. Any code in the method that follows the try – catch – finally will not be executed.

28. *Call Join after calling Abort if you need to wait for the thread to cleanup*
    When Abort is called on a Thread, ThreadAbortException is thrown (See item#27). All finally blocks are executed on the thread before the execution of the Thread terminates and this may take some time. The thread that calls the Abort may want to call Join right after calling Abort so it could wait for the Thread to terminate (See item#29 as well).

29. *One expects a Thread to terminate when Abort is called, do not ResetAbort*
    As discussed in item #27, a ThreadAbortException is automatically raised again at the end of the Catch to ensure termination of the Thread. However, in the catch block one may invoke ResetAbort to cancel the abort request and continue executing. In this case the thread will not terminate upon the Abort request. The calling code may not be aware of this and may result in undesirable behavior in the application. Consider redesigning if you think your application needs this feature.

30. *Use caution in Synchronizing on the Type object*
    Say you need to modify a static member of a class A, from within an instance method, in a thread safe manner. One possibility is to lock on the type object as:

    ```
    lock(GetType())
    ```

    This however may fail if we have two threads modifying the static members, where the first thread is invoking a method on the instance of the class and the second thread is invoking a method on an instance of its derived class. GetType will return two different Type objects; the Type of the class in first thread and the Type of the derived class in the second thread. The correct way to synchronize on the type object is:

    ```
    lock(typeof(A));
    ```

COM-Interop Gotchas

31. *ReleaseComObject to freeup unmanaged COM resources*
    Remember to call System.Runtime.InteropServices.Marshal.ReleaseComObject when you are done using a COM object from within a .NET application. If you do not call the ReleaseComObject, the COM object is not disposed until the Runtime Callable Wrapper is Finalized. You may end up holding critical resources for an extended period of time and this may affect the overall performance of the system.

32. *Caution using an interface after release*
    Say you have obtained multiple interfaces on a COM object. If you invoke ReleaseComObject using one of the interfaces, the Runtime Callable Wrapper is

disposed. Invoking methods now using any related interfaces will result in an object reference not set to an instance (NullReference) exception.

33. *Overhead associated with Apartments*
    If your COM object is in a STA (Single Threaded Apartment) and your .NET client is running in an STA as well. If a method is invoked on the COM component, only the COM interop overhead is incurred. However, if the .NET client were in an MTA, then in addition to the interop overhead, COM marshalling overhead is incurred as well since the request goes through a proxy and stub. It is important to find the apartment of the COM component and place the client in the same apartment, if possible, to eliminate the overhead.

34. *Know the default Apartment of your thread*
    What apartment does your .NET thread run on by default? The answer is: it depends! C# client runs in the MTA by default while VB.NET client runs in a STA by default.

35. *Do not rely on autogeneration of GUID*
    Set the GUIDAttribute on your class if you intend to make it available for COM interop. This provides greater control later on to vary the GUID if necessary.

36. *Set the ClassInterface Attribute to ClassInterfaceType.None*
    If you desire to interact with .NET from COM components/clients, do not let your .NET class expose any interfaces. Set the ClassInterfaceAttribute to ClassInterfaceType.None. Write a separate interface and expose its methods as interface for COM interop.

ASP.NET Gotchas

37. *Use caution with SmartNavigation*
    The SmartNavigation feature on an ASP.NET page has some neat features. However, when turned on, a request for the referrer (the referring page from which we arrived at this page) returns a null.

38. *Test for browser incompatibilities*
    While ASP.NET is pretty capable of generating proper output for different browsers, the scripts we write may not be. Test the application (all pages) in different browsers to ensure compatibility.

Web Services

39. *Set the Credential property of Proxy when invoking web service*
    On a Web Service client proxy set the Credentials property. This is more of an issue in development and testing. If web service is implemented on W2K, the default setting allows anonymous access. However on XP and Win2003, the Windows Integrated authentication is the default. Set the credentials to either DefaultCredential or to a NetworkCredential.

40. *Multiple Asynchronous calls using a single proxy may result in contention*
    The client proxy in .NET Web Services provides BeginInvoke and EndInvoke methods to emulate asynchronous calls. If you have a need to call multiple methods simultaneously on a web service, you may simply make these multiple asynchronous method calls using a single proxy. You will notice that simultaneous requests are being sent from the client and the response is indeed received simultaneously on the first set of calls. However, if the client proxy has the CookieContainer property set on it, the subsequent responses are received serially. The reason for this is that the same cookie is being transmitted on these multiple requests and the web server is serializing these calls based on the session. If you will use session state, use different proxies to send your simultaneous requests.

Conclusion

This article presents things that one should keep in mind while developing code on the .NET framework. .NET Framework provides a number of capabilities that makes a developer productive. However for its share, it has introduced a certain amount of complexity and issues as well. Knowing these Gotchas would allow us to fully benefit from the framework and language capabilities.

References

1. *Strings in Java and .NET*, http://www.agiledeveloper.com/download.aspx.
2. *Why copying an object is terrible thing to do?*, http://www.agiledeveloper.com/download.aspx.
3. http://msdn.microsoft.com.
4. Robert C Martin, *The Open-Closed Principle*. C++ Report, 1996. http://www.objectmentor.com/resources/articles/ocp.pdf.