Effective Software Development Series

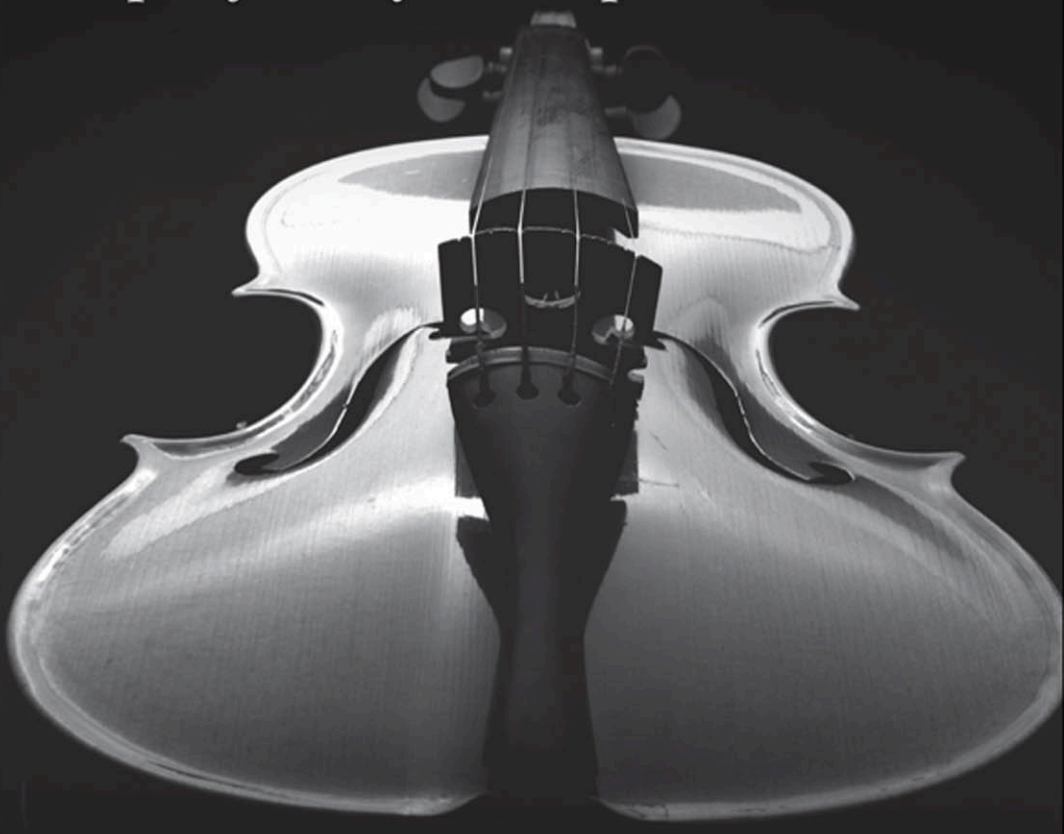
Scott Meyers, Consulting Editor

Effective C#

Third Edition

COVERS C# 6.0

50 Specific Ways to Improve Your C#



Content Update Program

FREE…See Details Inside

Bill Wagner

Effective C#

Third Edition

This book is part of InformIT's exciting new Content Update Program, which provides automatic content updates for major technology improvements!

- As significant updates are made to C#, this book will be updated or new sections will be added to match the updates to the technologies.
- The updates will be delivered to you via a free Web Edition of this book, which can be accessed with any Internet connection.
- This means your purchase is protected from immediately outdated information!

For more information on InformIT's Content Update program, see the inside back cover or go to **informit.com/cup**

If you have additional questions, please email our Customer Service department at ***informit@custhelp.com****.*

The sample compiles, because `B` objects aren't created, and any concrete derived object must supply an implementation for `VFunc()`. The C# strategy of calling the version of `VFunc()` matching the actual runtime type is the only possibility of getting anything except a runtime exception when an abstract function is called in a constructor. Experienced C++ programmers will recognize the potential runtime error if you use the same construct in that language. In C++, the call to `VFunc()` in the `B` constructor would crash.

Still, this simple example shows the pitfalls of the C# strategy. The `msg` variable is immutable. It should have the same value for the entire life of the object. Because of the small window of opportunity when the constructor has not yet finished its work, you can have different values for this variable: one set in the initializer, and one set in the body of the constructor. In the general case, any number of derived class variables may remain in the default state, as set by the initializer or by the system. They certainly don't have the values you thought, because your derived class's constructor has not executed.

Calling virtual functions in constructors makes your code extremely sensitive to the implementation details in derived classes. You can't control what derived classes do. Code that calls virtual functions in constructors is very brittle. The derived class must initialize all instance variables properly in variable initializers. That rules out quite a few objects: Most constructors take some parameters that are used to set the internal state properly. So you could say that calling a virtual function in a constructor mandates that all derived classes define a default constructor, and no other constructor. But that's a heavy burden to place on all derived classes. Do you really expect everyone who ever uses your code to play by those rules? I didn't think so. There is very little gain, and lots of possible future pain, from playing this game. In fact, this situation will work so rarely that it's included in the FxCop and Static Code Analyzer tools bundled with Visual Studio.

Item 17: Implement the Standard Dispose Pattern

We've discussed the importance of disposing of objects that hold unmanaged resources. Now it's time to cover how to write your own resource management code when you create types that contain resources other than memory. A standard pattern is used throughout the .NET Framework for disposing of unmanaged resources. The users of your type will expect you to follow this standard pattern. The standard dispose idiom frees

your unmanaged resources using the `IDisposable` interface when clients remember, and it uses the finalizer defensively when clients forget. It works with the garbage collector to ensure that your objects pay the performance penalty associated with finalizers only when necessary. This is the right way to handle unmanaged resources, so it pays to understand it thoroughly. In practice, unmanaged resources in .NET can be accessed through a class derived from `System.Runtime.Interop.SafeHandle`, which implements the pattern described here correctly.

The root base class in the class hierarchy should do the following:

- It should implement the `IDisposable` interface to free resources.
- It should add a finalizer as a defensive mechanism if and only if your class directly contains an unmanaged resource.
- Both `Dispose` and the finalizer (if present) delegate the work of freeing resources to a virtual method that derived classes can override for their own resource management needs.

The derived classes need to

- Override the virtual method only when the derived class must free its own resources
- Implement a finalizer if and only if one of its direct member fields is an unmanaged resource
- Remember to call the base class version of the function

To begin, your class must have a finalizer if and only if it directly contains unmanaged resources. You should not rely on clients to always call the `Dispose()` method. You'll leak resources when they forget. It's their fault for not calling `Dispose`, but you'll get the blame. The only way you can guarantee that unmanaged resources get freed properly is to create a finalizer. So if and only if your type contains an unmanaged resource, create a finalizer.

When the garbage collector runs, it immediately removes from memory any garbage objects that do not have finalizers. All objects that have finalizers remain in memory. These objects are added to a finalization queue, and the GC runs the finalizers on those objects. After the finalizer thread has finished its work, the garbage objects can usually be removed from memory. They are bumped up a generation because they survived collection. They are also marked as not needing finalization because the finalizers have run. They will be removed from memory on the next collection of that higher generation. Objects that need finalization stay in memory for far longer than objects without a finalizer. But you have no

choice. If you're going to be defensive, you must write a finalizer when your type holds unmanaged resources. But don't worry about performance just yet. The next steps ensure that it's easier for clients to avoid the performance penalty associated with finalization.

Implementing `IDisposable` is the standard way to inform users and the runtime system that your objects hold resources that must be released in a timely manner. The `IDisposable` interface contains just one method:

```
public interface IDisposable
{
    void Dispose();
}
```

The implementation of your `IDisposable.Dispose()` method is responsible for four tasks:

1. Freeing all unmanaged resources.
2. Freeing all managed resources (this includes unhooking events).
3. Setting a state flag to indicate that the object has been disposed of. You need to check this state and throw `ObjectDisposed` exceptions in your public members if any get called after disposing of an object.
4. Suppressing finalization. You call `GC.SuppressFinalize(this)` to accomplish this task.

You accomplish two things by implementing `IDisposable`: You provide the mechanism for clients to release all managed resources that you hold in a timely fashion, and you give clients a standard way to release all unmanaged resources. That's quite an improvement. After you've implemented `IDisposable` in your type, clients can avoid the finalization cost. Your class is a reasonably well-behaved member of the .NET community.

But there are still holes in the mechanism you've created. How does a derived class clean up its resources and still let a base class clean up as well? If derived classes override finalize or add their own implementation of `IDisposable`, those methods must call the base class; otherwise, the base class doesn't clean up properly. Also, finalize and `Dispose` share some of the same responsibilities; you have almost certainly duplicated code between the finalize method and the `Dispose` method. Overriding interface functions does not always work the way you'd expect. Interface functions are not virtual by default. We need to do a little more work to address these concerns. The third method in the standard dispose pattern, a protected virtual helper function, factors out these common tasks and adds a hook for derived classes to free resources they allocate. The

base class contains the code for the core interface. The virtual function provides the hook for derived classes to clean up resources in response to `Dispose()` or finalization:

```
protected virtual void Dispose(bool isDisposing)
```

This overloaded method does the work necessary to support both finalize and `Dispose`, and because it is virtual, it provides an entry point for all derived classes. Derived classes can override this method, provide the proper implementation to clean up their resources, and call the base class version. You clean up managed and unmanaged resources when `isDisposing` is `true`, and you clean up only unmanaged resources when `isDisposing` is `false`. In both cases, call the base class's `Dispose(bool)` method to let it clean up its own resources.

Here is a short sample that shows the framework of code you supply when you implement this pattern. The `MyResourceHog` class shows the code to implement `IDisposable` and create the virtual `Dispose` method:

```
public class MyResourceHog : IDisposable
{
    // Flag for already disposed
    private bool alreadyDisposed = false;

    // Implementation of IDisposable.
    // Call the virtual Dispose method.
    // Suppress Finalization.
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    // Virtual Dispose method
    protected virtual void Dispose(bool isDisposing)
    {
        // Don't dispose more than once.
        if (alreadyDisposed)
            return;
        if (isDisposing)
        {
            // elided: free managed resources here.
        }
```

```
        // elided: free unmanaged resources here.
        // Set disposed flag:
        alreadyDisposed = true;
    }

    public void ExampleMethod()
    {
        if (alreadyDisposed)
            throw new
                ObjectDisposedException("MyResourceHog",
                "Called Example Method on Disposed object");
        // remainder elided.
    }
}
```

If a derived class needs to perform additional cleanup, it implements the protected `Dispose` method:

```
public class DerivedResourceHog : MyResourceHog
{
    // Have its own disposed flag.
    private bool disposed = false;

    protected override void Dispose(bool isDisposing)
    {
        // Don't dispose more than once.
        if (disposed)
            return;
        if (isDisposing)
        {
            // TODO: free managed resources here.
        }
        // TODO: free unmanaged resources here.

        // Let the base class free its resources.
        // Base class is responsible for calling
        // GC.SuppressFinalize( )
        base.Dispose(isDisposing);

        // Set derived class disposed flag:
        disposed = true;
    }
}
```

Notice that both the base class and the derived class contain a flag for the disposed state of the object. This is purely defensive. Duplicating the flag encapsulates any possible mistakes made while disposing of an object to only the one type, not all types that make up an object.

You need to write `Dispose` and finalizers defensively. They must be idempotent. `Dispose()` may be called more than once, and the effect should be the same as calling them exactly once. Disposing of objects can happen in any order. You will encounter cases in which one of the member objects in your type is already disposed of before your `Dispose()` method gets called. You should not view that as a problem because the `Dispose()` method can be called multiple times. Note that `Dispose()` is the exception to the rule of throwing an `ObjectDisposedException` when public methods are called on an object that has been disposed of. If it's called on an object that has already been disposed of, it does nothing. Finalizers may run when references have been disposed of, or have never been initialized. Any object that you reference is still in memory, so you don't need to check null references. However, any object that you reference might be disposed of. It might also have already been finalized.

You'll notice that neither `MyResourceHog` nor `DerivedResourceHog` contains a finalizer. The example code I wrote does not directly contain any unmanaged resources. Therefore, a finalizer is not needed. That means the example code never calls `Dispose(false)`. That's the correct pattern. Unless your class directly contains unmanaged resources, you should not implement a finalizer. Only those classes that directly contain an unmanaged resource should implement the finalizer and add that overhead. Even if it's never called, the presence of a finalizer does introduce a rather large performance penalty for your types. Unless your type needs the finalizer, don't add it. However, you should still implement the pattern correctly so that if any derived classes do add unmanaged resources, they can add the finalizer and implement `Dispose(bool)` in such a way that unmanaged resources are handled correctly.

This brings me to the most important recommendation for any method associated with disposal or cleanup: You should be releasing resources only. Do not perform any other processing during a dispose method. You can introduce serious complications to object lifetimes by performing other processing in your `Dispose` or finalize methods. Objects are born when you construct them, and they die when the garbage collector reclaims them. You can consider them comatose when your program can no longer access them. If you can't reach an object, you can't call any