Username: Pralay Patoria **Book:** Visual Studio 2012 and .NET 4.5 Expert Development Cookbook. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Solutions to 10 common mistakes made by developers while writing code

Being a developer has always been a mammoth task to handle bad code. Most developers need to work in teams such that one's code needs to be recompiled and used by others. Mistakes in development by one of them adversely affects the entire application. Even most current programmers do not know the mistakes that they are making in the application. In this recipe, I am going to cover some of the interesting common mistakes that I see while working with developers and discuss their solutions.

Getting ready

We create a console application to demonstrate each case.

How to do it...

In this recipe, let's add some of the interesting mistakes which developers do, often either unknowingly or mistakenly. We will discuss each of the points individually:

1. Use of a destructor instead of `IDisposable`
2. Forgetting to call `Dispose` before going out of scope
3. Forgetting that strings are immutable
4. Breaking the call stack of an exception by throwing the exception again
5. Calling object members without making sure that the object cannot be null
6. Forgetting to unhook event handlers appropriately after wiring them
7. Not overriding `GetHashcode` when overriding `Equals`
8. Forgetting the call to the base constructor when calling derived objects
9. Putting large objects into static variables
10. Calling `GC.Collect` unnecessarily

How it works...

Let's see how we can solve the problems we described.

- Use of a destructor instead of `IDisposable`

  If you are declaring a class which uses references to unmanaged resources, it is important to dispose the object when it is going out of scope. In .NET you can use a destructor as its predecessor to release unmanaged resources which include file handles, database handles, and so on, such that the .NET environment calls the destructor automatically whenever GC gets executed. But this approach even though you don't need your caller to be aware of anything, is non-deterministic and adds some expenses. First of all, GC is non-deterministic, and hence the caller cannot make sure that the object has been disposed only after the object is dereferenced. The caller needs to wait for the GC to execute its finalizer until the resources are free from the object. Another important drawback is that the destructor loses the GC cycle, and hence the object will remain in memory for a longer time than expected. So if the class holds a reference to a large object, it might cause a delay in reclaiming the memory:

```
public class FinalizerClass
{
    public object LargeObject { get; set; }
    ~FinalizerClass()
    {
        //Release unmanaged references
        this.LargeObject = null;
    }
}
```

  In the preceding code, `LargeObject` is disposed from the finalizer thread, hence once the object is sent out of scope, it has to wait for GC to execute the destructor.

  .NET comes with a better approach with the disposable pattern, allowing the caller to release its resources. Thus, the caller knows when to release the resources and this approach works in a deterministic manner:

```
public class DisposableClass : IDisposable
{
    public object LargeObject { get; set; }
    public void Dispose()
    {
        //Release unmanaged references
        this.LargeObject = null;
    }
}
```

Here, the caller needs to call `Dispose` whenever it wants to release the resources from memory. The disposable pattern does not lose the GC cycle and hence works best in a managed environment.

- Forgetting to call `Dispose` before going out of scope

  Just similar to previous example, when you are using a disposable object, it is important to call the `Dispose` method before the object goes out of scope to release unnecessary resources from memory. C# comes with a shortcut to ensure that the object calls the `Dispose` method when going out of scope. It is always a good idea to use the `using` block when using a disposable object to avoid the problem:

```
public void CallDisposable()
{
    using (DisposableClass dclass = new DisposableClass())
    {
        //Working with the dclass
    }
}
```

  In the preceding code, the `Dispose` method will automatically be called when `dclass` is going out of scope of the `using` block.

- Forgetting that strings are immutable

  In .NET, strings are immutable. So when you do a slight manipulation on a `String` element, the entire memory is recreated again to store the final result. The value of a string once created cannot be changed. For instance:

```
public void StringWhoes()
{
    string thisString = "This " + "is " + "a " + "string.";

    thisString.Replace("is", "was");

    Console.Write(thisString); //wrong

    thisString = thisString.Replace("is", "was");

    Console.Write(thisString);
}
```

  In the preceding code, we declared a string dynamically. It is important to note that (even though here the IL optimization does not affect any performance) strings are immutable, hence for each string concatenation, a new memory is created. It is better to use `StringBuilder` or `string.Format` rather than `String` when we need to append strings with some other variables. Another interesting fact is that as strings are immutable, you cannot replace the object string using the `Replace` method. You need to store the result into another reference to get the result from `Replace` .

- Breaking the call stack of an exception by throwing the exception again

  Breaking the call stack of an exception is another important mistake that developers often make. Let us look at the following code:

```
public void BreakCallStack()
{
    try
    {
        this.Call1();
    }
    catch (Exception ex)
    {
        throw ex;       // Wrong
        throw new ApplicationException("Exception occurred",
ex); // wrong

        throw;          // right
    }
}

private void Call1()
{
    this.Call2();
}

private void Call2()
{
    throw new NotImplementedException();
}
```

In the preceding code, `BreakCallStack` calls a method `Call1` which in turn calls `Call2`. Now when the exception is caught in `BreakCallStack`, it will hold the information of the entire stack in `ex`. Calling `throw ex`, or wrapping the exception into another exception will eventually lose the stack. So rather than using the first two constructs it is good to use the third construct (`throw`) to retain the call stack.

- Calling object members without making sure that the object cannot be null

  `NullReferenceException` is one of the most common forms of exception that we see regularly. While developing code, it is always important to check every parameter that is coming as an argument, or any external object to null before calling its members:

```
public string CallWithNull(object param1)
{
    return param1.ToString();
}
```

  In the preceding code the method is called with a parameter `param1` of which we pass the `ToString` implementation. Now if the `param1` parameter holds null during any call, the method will produce `NullReferenceException`. Hence, it is important to check if `param1` equals `null` before invoking any method.

- Forgetting to unhook event handlers appropriately after wiring them

  A memory leak can occur when an event is hooked to an event source but never unhooked. Generally, an event takes a delegate as argument, which is passed from the caller. But if you do not unhook the event handler from the actual object where the event exists, the object will not be garbage collected until the object that holds the event handler gets exposed to GC. Events in .NET holds a strong reference to `EventSource`, and hence it is very important to unhook the event handlers when not in use.

- Not overriding `GetHashcode` when overriding `Equals`

  Sometimes developers forget to override `GetHashCode` even after the overriding equals. When we equate two objects, the .NET first calls `GetHashCode` of each of the objects before calling the `Equals` method. So if the hash code of two objects does not match, they will never be considered as equal. Hence, you need to override `GetHashCode` of both the types and return the same value for two objects.

- Forgetting the call to the base constructor when calling derived objects

  Sometimes we forget to call the base object from the constructor of a derived object. It is important to note that we must construct the base object totally before doing the work in the derived object. For instance:

```
public class DerivedNode : BaseNode
{
    public DerivedNode(string derivednote) //wrong
    {
        this.DerivedNote = derivednote;
    }
    public DerivedNode(string basenote, string derivednote) //
right
        : base(basenote)
    {
        this.DerivedNote = derivednote;
    }

    public string DerivedNote { get; set; }
}
```

  In the preceding code, the first constructor does not call the base constructor but rather calls the default constructor that does nothing, whereas the second constructor actually creates the base object completely.

- Putting large objects into static variables

  Static variables are disposed once `AppDomain` is unloaded. Hence, when you are using a static object, you need to remember that those are long living objects and will never be reclaimed during the execution of the process until `AppDomain` is unloaded (which is a rare scenario for a program). Hence, putting a large object in a static variable will put additional pressure on the process memory. Another important consideration that you need to remember is that calling a static member from multiple threads can create a nightmare. You must make static members thread safe when being accessed from concurrent threads.

- Calling GC.Collect unnecessarily

  Developers are often prone to call `GC.Collect` to invoke the GC cycle. Generally, GC collection blocks the execution engine if it is not a background GC and hence, can create performance bottlenecks in the application. GC collection is automatically managed by the CLR, and hence it is recommended to avoid the explicit call.

There's more...

Even though we have already discussed the 10 common mistakes that developers often do while developing an application, it is not an entire list. Let's identify some of the additional mistakes that can be taken care of.

Never declare structs if you don't have good reason to

I have seen developers declare structs in their code when they do not need member functions or inheritance. Generally, it is important to note that `struct` even though it exists in languages, should be avoided. Generally, the limit of `struct` is 16 bytes, if it is anything over that threshold, the performance of `struct` degrades from a class and it is better to use a class instead of `struct` .

Another interesting fact is that, a struct will automatically allocate itself whenever the object is declared, and you cannot put your custom logic into the default constructor of a struct. Structs should always be immutable if it is declared.

Do not create a list from IEnumerable before iterating on it

`IEnumerable` is a special interface that implements a state machine to generate the sequence of objects. It is important to note that sometimes when we need to directly call one object with its index, you need to create a deterministic list of objects from the `IEnumerable` list. But you should always remember, `IEnumerable` is good at iterating. So, if you do not need a cached implementation, and just need to use the list to iterate on an object, it is not good to convert `IEnumerable` to a list.

For instance, consider the following code:

```
public void GetList(IEnumerable<string> strings)
{
    List<string> lstrings = strings.ToList();
    foreach (string lstring in lstrings)
    {
        //Write custom logic
        }
    }
```

In the preceding code, it is not necessary to create a list of strings before going to the `foreach` loop. The `ToList` extension method actually generates the list by iterating using `foreach` and putting it into the cached list. So here, the performance is degraded because the same sequence is iterated twice.

See also…

- See http://bit.ly/ProgrammingMistakes