Visiting Files and Directories in C#

Recently I have been doing a lot of work with continuous integration [CI] using CruiseControl.Net [CCNet]. It is currently an adequate product, but has the potential to be an excellent product. I'm looking forward to the next release, which is likely to be the first for .Net 2.

One particular requirement I had was to remove the source tree from the disk after a change has been identified and before the change is checked out. This effectively gives the user a completely clean check out each time. This feature is not supported by any of the current CruiseControl.Net source control tasks. It is possible to use a pre-build task to make a system call to remove the source, but this is a hack and not portable easily across platforms.

In my article Continuous Integration with CruiseControl.Net – Part 3 (available soon from http://www.marauder-consulting.co.uk) I wrote about the powerful plug-in mechanism supported by CruiseControl.Net. It can be used to modify existing task blocks to add, for example, the ability to remove a source tree prior to checkout. C# is the ideal language to use to write CruiseControl.Net plug-ins as they are simply .Net assemblies.

In this article I am going to look at how to use C# to remove a source tree and develop the code into a enumeration method [EnumMethod] and visitor [Visitor] compound that can be used for general purpose file and directory traversal.

`Directory.Delete`

The .Net directory class has a `Delete` static method that takes a path to the directory to be deleted. At first glance it appears to be the answer to the problem and could make this article very short indeed (in reality all it did was make my estimate of the time it would to write the plug-in wildly inaccurate). To test it I created a temporary folder and put an empty text file in it, then wrote the following code:

```
Directory.Delete(@"C:\temp\somedir");
```

I was surprised to find that an exception was thrown. When I added exception handling code the message was as follows:

```
The directory is not empty.
```

This seemed fair enough and the `Directory.Delete` method does have an overload that takes a second parameter called `recursive`, which claims to remove all files and directories in the path recursively. I modified the code and tried again:

```
Directory.Delete(@"C:\temp\somedir",true);
```

Success! The file and the directory were both deleted. For good measure (although now it just feels like proving black was white and getting killed on the next zebra crossing [HHGTTG]) I tried the same code on a source tree. It failed with the error message:

```
Access to the path 'all-wcprops' is denied.
```

Some files had been deleted, but others had not. So I was left wondering what was special about the files that hadn't been deleted.

I have only worked with a few source control systems, namely CVS [CVS], SubVersion [SVN] and Perforce [P4]. Perforce and at least one configuration of CVS that I have used checks all its files out as read-only. Files in the special `.SVN` folders used by SVN to keep track of changes in the source are also read-only. `'all-wcprops'` is one of these files. I started wondering if the problem could be caused by read-only files, so I recreated my directory an empty text file, this time making it read-only. I ran the code again and got the same error, this time relating to the file in my directory.

This suggested that the solution was to remove the read-only flag from files before deleting them. The `FileAttributes` enum and `File` class can be used to test and set the attributes of a file, including read-only:

```
string file = @"C:\temp\somedir\somefile.txt";
FileAttributes fileAtts = File.GetAttributes(file);
File.SetAttributes(file, fileAtts & ~FileAttributes.ReadOnly);

Directory.Delete(@"C:\temp\somedir",true);
```

Executing the above code removed the file and the directory. However, I could not test it on the source tree as that would require a list of all the files in the tree that were read-only.

Listing Read-Only Files in a Directory

Listing the files in a directory in .Net is simple. All you need is a `DirectoryInfo` object for the directory and then you can iterate through a collection returned by the `GetFiles` method. To test this I added some more files to the test directory, made three of them read-only and then ran the following code:

```
DirectoryInfo dirInfo = new DirectoryInfo(@"C:\temp\somedir");

foreach (FileInfo fileInfo in dirInfo.GetFiles())
{
      FileAttributes fileAtts =
            File.GetAttributes(fileInfo.FullName);
      if ((fileAtts & FileAttributes.ReadOnly) != 0)
      {
            Console.WriteLine(fileInfo.FullName);
      }
}
```

As expected the output listed the read-only files. The next step was to remove the read-only attribute from the files. It also makes sense to remove the files at the same time as that is the ultimate goal. The `File.Delete` static method is used for this:

```
DirectoryInfo dirInfo = new DirectoryInfo(@"C:\temp\somedir");

foreach (FileInfo fileInfo in dirInfo.GetFiles())
{
      FileAttributes fileAtts =
            File.GetAttributes(fileInfo.FullName);
      if ((fileAtts & FileAttributes.ReadOnly) != 0)
      {
            File.SetAttributes(fileInfo.FullName,
                        fileAtts & ~FileAttributes.ReadOnly);
      }

      File.Delete(fileInfo.FullName);
}
```

This is still not a complete solution as a source tree will also contain subdirectories that have some read-only files.

Listing Subdirectories

A directory's subdirectories can be listed in much the same way as its files, but instead of using the `GetFiles` method the `GetDirectories` method is used:

```
DirectoryInfo dirInfo = new DirectoryInfo(@"C:\temp\somedir");

foreach (DirectoryInfo subDirInfo in dirInfo.GetDirectories())
{
      Console.WriteLine(subDirInfo.FullName);
}
```

I tested this in the obvious way, by creating some subdirectories in my test directory. The above code listed them all.

Putting it All Together – Recursion

I now had a method of:

- Listing all files in a directory
- Detecting if a file is read-only
- Removing the read-only attribute from a file
- Deleting a file
- Listing all subdirectories in a directory

The final solution needs to:

- Visit every directory in the source tree
- Remove the directory's files and subdirectories
- Remove the directory itself.

The easiest way to do this is using recursion [Recursion]:

```
private static void DeleteDirectory(string path)
{
      DirectoryInfo dirInfo = new DirectoryInfo(path);

      foreach (DirectoryInfo subDirInfo in
                        dirInfo.GetDirectories())
      {
            DeleteDirectory(subDirInfo.FullName);
      }

      foreach (FileInfo fileInfo in dirInfo.GetFiles())
      {
            FileAttributes fileAtts =
                  File.GetAttributes(fileInfo.FullName);
            if ((fileAtts & FileAttributes.ReadOnly) != 0)
            {
                  File.SetAttributes(fileInfo.FullName,
                        fileAtts & ~FileAttributes.ReadOnly);
            }

            File.Delete(fileInfo.FullName);
      }

      Directory.Delete(path);
}
```

In the example above `DeleteDirectory` is called recursively for every directory in the supplied path until a directory is reached that doesn't have any subdirectories or all subdirectories have been visited. Then it works back up the tree of directories removing the read-only attribute from read-only files and deleting all files.

This of course is the solution to the problem. I ran it on the source tree and it deleted all files and directories. So this article is finished, right?

The Patterns

Wrong! The `DeleteDirectory` method is a very specific solution to a very specific problem. It can not be used for doing anything other than deleting directories and their contents. However, wanting to *visit* and operate on every file and directory in a tree is a common requirement. Imagine, for example, you were writing an application similar to Windows explorer or any other application that requires a list of files and directories.

The functionality provided by `DeleteDirectory` can be separated out and expressed as two patterns. Iterating through directories and files is an implementation of the Enumeration Method pattern and operating on each file and directory is a (well hidden at the moment) implementation of the Visitor pattern.

`DirectoryTraverser`

Let's start by looking at the class that will traverse through the directories and files. `DeleteDirectory` works very well as simple static method, but with `DirectoryTraverser` we want to be able to plug-in different functionality and the power that a class provides makes life very much easier, especially in terms of interface.

The `DirectoryTraverser` class only needs a single, non-constructor, public method in its interface. This method is used to specify which directory to traverse and to start the traversal:

```
class DirectoryTraverser
{
    public void Traverse(string path)
    {
       …
    }
}
```

`DirectoryInfo` and `FileInfo` objects are used when the actual file and directory traversal is performed. When the `DeleteDirectory` method is called recursively the full path to the next directory is extracted from the `DirectoryInfo` object. This is an unnecessary property access. This can be avoided by providing a private overload of `Traverse` that takes a `DirectoryInfo` object and use the original method to create the `DirectoryInfo` object to be passed the first time the overload is called.

```
class DirectoryTraverser
{
    public void Traverse(string path)
    {
        Traverse(new DirectoryInfo(path));
    }

    private void Traverse(DirectoryInfo dirInfo)
    {
       …
    }
}
```

If were writing an application that sends an indented list of directories and files to the console, we want to know about a directory before we know about its files. If we're writing an application that deletes directories and files we want to know about each file before the directory so that we can delete the files and then delete the directory. Order is very important. Therefore we need two directory related actions into which we can plug functionality. One invoked when entering a directory. The other invoked when leaving a directory. Only a single pluggable action is needed for files:

```
private void Traverse(DirectoryInfo dirInfo)
{
    EnterDirectory(dirInfo);

    foreach (DirectoryInfo subDir in dirInfo.GetDirectories())
    {
        Traverse(subDir);
    }

    foreach (FileInfo file in dirInfo.GetFiles())
    {
        VisitFile(file);
    }

    LeaveDirectory(dirInfo);
}

private void EnterDirectory(DirectoryInfo dirInfo)
{
}

private void VisitFile(FileInfo fileInfo)
{
}

private void LeaveDirectory(DirectoryInfo dirInfo)
{
}
```

That completes *most* of the functionality of `DirectoryTraverser`. However, at the moment it is not very pluggable.

`IDirectoryVisitor`

The visitor part of the implementation is the ability to plug-in behaviour when a directory is entered, when a file is visited and when a directory is left. The easiest way to do this is via a method call for each operation and C#'s delegate mechanism would seem to be the obvious implementation solution. In fact my early implementations used it. It is a good solution if you only have one method to delegate. As soon as you have more it becomes cumbersome. Do you set the delegates via the constructor or properties or both? What if you only want a delegate for entering directories and visiting files? Do you have a constructor that takes only two delegates or only one? If so which one? Which two? etc.

The simplest solution in this case is to have a callback interface, such as the one below:

```
interface IDirectoryVisitor
{
    void EnterDirectory(DirectoryInfo dirInfo);

    void VisitFile(FileInfo fileInfo);

    void LeaveDirectory(DirectoryInfo dirInfo);
}
```

Then you can just implement the methods you want and leave the others as empty methods that do nothing. For example the following implementation of the interface lists all directories and files in the given path.

```
class DirectoryWriter : IDirectoryVisitor
{
    public void EnterDirectory(DirectoryInfo dirInfo)
    {
        Console.WriteLine(dirInfo.FullName);
    }

    public void VisitFile(FileInfo fileInfo)
    {
        Console.WriteLine(fileInfo.FullName);
    }

    public void LeaveDirectory(DirectoryInfo dirInfo)
    {
    }
}
```

Now all that is needed is a method of plugging an implementation of the `IDirectoryVisitor` into the `DirectoryTraverser`. This is easy and straight forward. Simply pass an interface reference into the constructor, store and call the appropriate callback, instead of the methods show previous:

```
class DirectoryTraverser
{
    private IDirectoryVisitor visitor;

    public DirectoryTraverser(IDirectoryVisitor visitor)
    {
        this.visitor = visitor;
    }

    public void Traverse(string path)
    {
        Traverse(new DirectoryInfo(path));
    }

    private void Traverse(DirectoryInfo dirInfo)
    {
        visitor.EnterDirectory(dirInfo);

        foreach (DirectoryInfo subDir in
                     dirInfo.GetDirectories())
        {
            Traverse(subDir);
        }

        foreach (FileInfo file in dirInfo.GetFiles())
        {
            visitor.VisitFile(fileInfo);
        }

        visitor.LeaveDirectory(dirInfo);
    }

}
```

The class above does of course have a, strictly speaking, extra level of unnecessary indirection: the calls to the `EnterDirectory`, `VisitFile` and `LeaveDirectory` methods could be replaced with calls directly to the visitor.

`DirectoryTraverser` and `DirectoryWriter` are used together as follows:

```
DirectoryTraverser dirTrav =
                 new DirectoryTraverser( new DirectoryWriter());
dirTrav.Traverse(@"C:\temp");
```

Coming Full Circle

The `DirectoryWriter` visitor above is very simple, not especially useful and does not demonstrate how the `EnterDirectory` and `LeaveDirectory` callbacks can be used together. The more advanced version shown below uses the methods to control the indentation when sending the directory and file structure to the console:

```
class DirectoryWriter : IDirectoryVisitor
{
    private int indent = 0;

    public void EnterDirectory(DirectoryInfo dirInfo)
    {
        StringBuilder buffer = new StringBuilder();
        buffer.Append( new string( ' ', indent ) );
        buffer.Append(dirInfo.Name);
        Console.WriteLine(buffer.ToString());
        ++indent;
    }

    public void VisitFile(FileInfo fileInfo)
    {
        StringBuilder buffer = new StringBuilder();
        buffer.Append(new string(' ', indent+1));
        buffer.Append(fileInfo.Name);
        Console.WriteLine(buffer.ToString());
    }

    public void LeaveDirectory(DirectoryInfo dirInfo)
    {
        --indent;
    }
```

Running it against the test project I created for this article gives the following output:

```
DirectoryTraverser2
 DirectoryTraverser2
  bin
   Debug
     DirectoryTraverser2.exe
     DirectoryTraverser2.pdb
     DirectoryTraverser2.vshost.exe
   Release
     DirectoryTraverser2.exe
     DirectoryTraverser2.pdb
  obj
   Debug
    TempPE
     DirectoryTraverser2.exe
     DirectoryTraverser2.pdb
   Release
    TempPE
     DirectoryTraverser2.exe
     DirectoryTraverser2.pdb
    DirectoryTraverser2.csproj.FileList.txt
  Properties
    AssemblyInfo.cs
   DirectoryTraverser.cs
   DirectoryTraverser2.csproj
   DirectoryWriter.cs
   EntryPoint.cs
   IDirectoryVisitor.cs
  DirectoryTraverser2.sln
```

An example of a visitor that deletes files and directories is shown below to bring this article full circle.

```
class DeleteDirectoryVisitor : IDirectoryVisitor
{
    public void EnterDirectory(DirectoryInfo dirInfo)
    {
    }

    public void VisitFile(FileInfo fileInfo)
    {
        FileAttributes fileAtts =
                         File.GetAttributes(fileInfo.FullName);
        if ((fileAtts & FileAttributes.ReadOnly) != 0)
        {
            File.SetAttributes(fileInfo.FullName,fileAtts &
                         ~FileAttributes.ReadOnly);
        }

        File.Delete(fileInfo.FullName);
    }

    public void LeaveDirectory(DirectoryInfo dirInfo)
    {
        Directory.Delete(dirInfo.FullName);
    }
}
```

Acknowledgments

Thank you to Kevlin Henney for reviews and guidance.

References

[CI] http://en.wikipedia.org/wiki/Continuous_integration
[CCNet] http://ccnet.thoughtworks.com/
[EnumMethod] http://www.two-sdg.demon.co.uk/curbralan/papers/ATaleOfThreePatterns.pdf
[Visitor] Design patterns: elements of reusable object-oriented software by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. ISBN-10: 0201633612 ISBN-13: 978-0201633610
[HHGTTG] http://www.bbc.co.uk/cult/hitchhikers/
[CVS] http://www.nongnu.org/cvs/
[SVN] http://subversion.tigris.org/
[P4] http://www.perforce.com/
[Recursion] http://en.wikipedia.org/wiki/Recursion
[GoF] Design patterns : elements of reusable object-oriented software by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. ISBN-10: 0201633612 ISBN-13: 978-0201633610