

## Testing Visiting Files and Directories in C#

### Visiting Files and Directories in C#

In my previous article, Visiting Files and Directories in C# [VFDC#], I looked at how to use C# to remove a source tree and developed the code into an enumeration method [EnumMethod] and visitor [Visitor] compound that can be used for general purpose file and directory traversal:

```
public interface IDirectoryVisitor
{
    void EnterDirectory(DirectoryInfo dirInfo);
    void VisitFile(FileInfo fileInfo);
    void LeaveDirectory(DirectoryInfo dirInfo);
}

public 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(file);
        }

        visitor.LeaveDirectory(dirInfo);
    }
}
```

The article did not discuss any form of automated testing. This not only makes me very uncomfortable, it also means the classes cannot be modified or refactored safely. In this article I am going to look at how to write automated tests for `DirectoryTraverser` and discuss the differences between unit and integration testing and when to use them.

## Unit and Integration Testing

The message about automated testing is finally getting through. However, many organisations are still not doing it. The majority of those that are using automated tests cannot see past unit testing or their unit tests are a mixture of unit tests and integration tests. I'm going to use an example to demonstrate the difference between a unit test and an integration test and explain when each should be used.

Imagine you have a class, called `HistoricPrices`, that is used to retrieve historic stock prices from a database. The class constructor takes an `IDBConnection` interface that is used to make direct calls to a database to retrieve the prices. A test is run every time the project containing `HistoricPrices` is compiled by the developer on their local machine. The developer needs the test to run quickly and give accurate repeatable results, so instead of passing in a real `DBConnection` (the class that implements `IDBConnection` in production) object they pass in a fake [Fake]. The fake object has a number of hard-coded recordsets that are mapped to a set of predetermined SQL strings. This means that every time a particular SQL string is executed via the `IDBConnection` interface to the fake object, the same recordset is always returned. As the fake object does not actually talk to the database the recordset is returned in (almost) zero time. This makes the tests very fast and easy to run. This is a *unit* test.

The unit test only tests a very small part of the system, in this case just one class. Every system should be tested as fully as possible. In the case of `HistoricPrices` the interaction with a real database is vital and should also be tested. The developer still requires the tests to give accurate repeatable results, so instead of using a live database a test database is constructed, tested against using the production `DBConnection` object and torn down every time the test is run. The test takes some time to run and tests the interaction between a `HistoricPrices` object and a real database. This is an *integration* test.

In my experience, creating and dropping a SQL Server database on a developer spec machine can take anything up to 20 seconds. That is a long time to wait, so the developer is likely to be less keen to run the test every time they compile. Therefore the integration test should be run, at the very least, prior to a release and ideally prior to checkin, as part of a nightly build of the entire system and/or as part of continuous integration [CI].

So, to recap, unit testing is about removing dependencies and writing tests that run in (almost) zero time, so that they can be run every time the compiler is invoked. Integration tests test the interaction between at least two things. For example between two objects or between an object and a database or an object and a file. This means that integration tests potentially take a while to run and traditionally this puts developers off running them every time they invoke the compiler.

## Unit Testing DirectoryTraverser

The only parts of the .Net library that intrude into `DirectoryTraverser` are `DirectoryInfo` and `FileInfo`. So all that should be needed to be done to write a unit test is to mock these out and create a suitable factory to create the mocks when testing and the real objects when in production. The problem is that `DirectoryInfo` and `FileInfo` do not already have suitable interfaces, so new interfaces must be written and the original classes wrapped. That in itself is not too much trouble. Unfortunately `GetDirectories` and `GetFiles` methods return a `DirectoryInfo[]` and `FileInfo[]` respectively and therefore their return values must be mapped onto arrays of the new interface types. Suddenly you have much more test code than code being tested and it is far more complex, so a unit test is not appropriate or worthwhile in this case.

## Unit Testing DirectoryTraverser

As previously stated, integration testing is the testing of how one or more units or modules work together. In the case of `DirectoryTraverser` we need to test how it integrates with the file system. This involves creating a known set of directories and files, traversing them, checking the results and, of course, cleaning up afterwards.

## Creating Directories and Files

Before we consider how to create directories and files we must consider *where* to create them. It needs to be a place where the following test code can find them and where they won't interfere with anything else in the file system. We could just pick a path, such as `c:\temp`, but that would only work on Windows machines. .Net has the ideal solution:

```
Path.GetTempPath()
```

The `GetTempPath` method returns the path to a directory that can be used to store temporary files and directories. The path is specific to the operating system in use. So on Windows it's something along the lines of:

```
C:\Documents and Settings\user\Local Settings\Temp
```

And on Linux it's along the lines of:

```
/home/user/tmp
```

Now we're ready to create the directories and files. .Net makes this very easy as both the `Directory` and `File` classes have create methods that take a string:

```
// Creating a directory.
Directory.CreateDirectory("...");

// Creating a file
FileStream str = File.Create("...");
str.Close();
```

The only thing to remember is that the `File.Create` method returns an open `FileStream` and must be closed so the file can be accessed in the test, as you cannot rely on `Dispose` being called in time.

As well as using different temporary paths, different operating systems also use different directory separators. The `Path.Combine` method will concatenate *two* strings together with the correct separator for the operating system. So, for the path

```
test\dir1\dir2
```

you would need to write:

```
string fullPath = "test";
fullPath = Path.Combine(fullPath, "dir1" );
fullPath = Path.Combine(fullPath, "dir2" );
```

This is more than a little tedious, especially for long or multiple paths, and is not especially clear. It would be much nicer to be able to write:

```
string fullPath = MakePath("test", "dir1", "dir2");
```

The C# `params` keyword allows methods to take a varying number of arguments and access them as an array, which in turn allows us to write:

```
static private string MakePath(params string[] tokens)
{
    string fullpath = "";
    foreach (string token in tokens)
    {
        fullpath = Path.Combine(fullpath, token);
    }
    return fullpath;
}
```

All that's left is to define the directories and file we want to create. This is easily and clearly done using arrays:

```
string testFolderPath = Path.GetTempPath();

string[] testDirs = {
    MakePath(testFolderPath, "Test"),
    MakePath(testFolderPath, "Test", "dir1"),
    MakePath(testFolderPath, "Test", "dir1", "dir2" ),
    MakePath(testFolderPath, "Test", "dir1", "dir2", "dir3") };

string[] testFiles = {
    MakePath(testFolderPath, "Test", "dir1", "file1.txt"),
    MakePath(testFolderPath, "Test", "dir1", "file2.txt"),
    MakePath(testFolderPath, "Test", "dir1", "dir2", "file3.txt"),
    MakePath(testFolderPath, "Test", "dir1", "dir2", "file4.txt") };
```

I have chosen a very simple directory structure: A root directory called `Test` with two subdirectories, `dir1` and `dir2`, each containing two text files `file1.txt`, `file2.txt`, `file3.txt` and `file4.txt` and a further empty subdirectory, `dir3`. This allows us to test that `DirectoryTraverser`:

1. Enters and leaves directories in sequence.
2. Visits all files.
3. Visits all subdirectories.
4. Empty directories are handled correctly.

Creating the files and directories is easily accomplished using a couple of `foreach`'s:

```
// Create directories
foreach (string dir in testDirs)
{
    Directory.CreateDirectory(dir);
}

// Create files
foreach (string file in testFiles)
{
    FileStream str = File.Create(file);
    str.Close();
}
```

The directories and files should be removed after the test. This can be achieved using the `Directory.Delete` method and setting the recursive flag (see `Visiting Files and Directories in C#`):

```
Directory.Delete(testFolderPath + "Test", true);
```

Finally the create and delete code needs to be put into the `SetUp` and `TearDown` methods of an NUnit [NUnit] test fixture:

```
[TestFixture]
public class DirectoryTraverserTest
{
    private readonly string testFolderPath = Path.GetTempPath();

    static private string MakePath(params string[] tokens)
    {
        string fullpath = "";
        foreach (string token in tokens)
        {
            fullpath = Path.Combine(fullpath, token);
        }
        return fullpath;
    }

    [SetUp]
    public void Setup()
    {
        Directory.CreateDirectory(testFolderPath);

        string[] testDirs = {
            MakePath(testFolderPath, "Test"),
            MakePath(testFolderPath, "Test", "dir1"),
            MakePath(testFolderPath, "Test", "dir1", "dir2"),
            MakePath(testFolderPath, "Test", "dir1", "dir2", "dir3") };

        foreach (string dir in testDirs)
        {
            Directory.CreateDirectory(dir);
        }

        string[] testFiles = {
            MakePath(testFolderPath, "Test", "dir1", "file1.txt"),
            MakePath(testFolderPath, "Test", "dir1", "file2.txt"),
            MakePath(testFolderPath, "Test", "dir1", "dir2", "file3.txt"),
            MakePath(testFolderPath, "Test", "dir1", "dir2", "file4.txt") };

        foreach (string file in testFiles)
        {
            {
                FileStream str = File.Create(file);
                str.Close();
            }
        }

        [TearDown]
        public void TearDown()
        {
            Directory.Delete(testFolderPath, true);
        }
    }
}
```

This is the best of many options I considered for creating the directories and files. Other options included:

- Traversing XML to get the structure.
- Storing the structure in a zip file that would be extracted each time the test was run.
- Writing the structure to an output file and using an external tool for test verification.

The advantage of the final solution is that it is simple and all in the code with no need for an external XML file, zip file or external tool.

## Test Visitor

DirectoryTraverser won't do anything without a visitor. Of the four tests listed in the previous section, 1 is the easiest to implement. All that is needed is a stack. When EnterDirectory is called the directory path is pushed onto the stack. When LeaveDirectory is called, a path is popped from the stack and compared to the path of the directory just left. As long as they are the same the test passes:

```
class DirRecorder : IDirectoryVisitor
{
    private Stack<string> lastDir = new Stack<string>();

    public void EnterDirectory(DirectoryInfo dirInfo)
    {
        lastDir.Push(dirInfo.FullName);
    }

    public void VisitFile(FileInfo fileInfo)
    {
    }

    public void LeaveDirectory(DirectoryInfo dirInfo)
    {
        Assert.AreEqual(lastDir.Pop(), dirInfo.FullName);
    }
};
```

To run the test, an instance of the visitor must be created and passed to an instance of DirectoryTraverser. Then the DirectoryTraverser instance must be passed the path to traverse:

```
[Test]
public void TraverseDirectory()
{
    string testPath = Path.Combine(testFolderPath, "Test");

    DirRecorder dirRecorder = new DirRecorder();
    DirectoryTraverser trav = new DirectoryTraverser(dirRecorder);
    trav.Traverse(testPath);
}
```

The easiest way to ensure that all file and directories are entered and all files are visited is to create a list of both and compare them to lists of expected directories and files. The order in which directories are entered and files are visited is not guaranteed, so all lists must be sorted. The expected lists can be generated at the same time as the physical directories and files are created (the highlighted code shows the modifications):



```
[TestFixture]
public class DirectoryTraverserTest
{
    private List<string> expectedDirs = new List<string>();
    private List<string> expectedFiles = new List<string>();

    ...

    [SetUp]
    public void Setup()
    {
        ...

        foreach (string dir in testDirs)
        {
            expectedDirs.Add(dir);
            Directory.CreateDirectory(dir);
        }
        expectedDirs.Sort();

        ...

        foreach (string file in testFiles)
        {
            expectedFiles.Add(file);
            FileStream str = File.Create(file);
            str.Close();
        }
        expectedFiles.Sort();
    }

    ...
}
```

The visitor can be modified to keep a list of entered directories and visited files, and accessors provided to retrieve the lists:

```
class DirRecorder : IDirectoryVisitor
{
    private List<string> dirs = new List<string>();
    private List<string> files = new List<string>();
    private Stack<string> lastDir = new Stack<string>();

    public List<string> Dirs
    {
        get
        {
            dirs.Sort();
            return dirs;
        }
    }
}
```

```
public List<string> Files
{
    get
    {
        files.Sort();
        return files;
    }
}

public void EnterDirectory(DirectoryInfo dirInfo)
{
    dirs.Add(dirInfo.FullName);
    lastDir.Push(dirInfo.FullName);
}

public void VisitFile(FileInfo fileInfo)
{
    files.Add(fileInfo.FullName);
}

public void LeaveDirectory(DirectoryInfo dirInfo)
{
    Assert.AreEqual(lastDir.Pop(), dirInfo.FullName);
}
};
```

Then the `TraverseDirectory` test can be modified to compare the lists of visited directories and files with the expected lists.

```
[TestFixture]
public class DirectoryTraverserTest
{
    private readonly string testFolderPath = Path.GetTempPath();
    private List<string> expectedDirs = new List<string>();
    private List<string> expectedFiles = new List<string>();

    static private string MakePath(params string[] tokens)
    {
        string fullpath = "";
        foreach (string token in tokens)
        {
            fullpath = Path.Combine(fullpath, token);
        }
        return fullpath;
    }

    [SetUp]
    public void Setup()
    {
        Directory.CreateDirectory(testFolderPath);

        string[] testDirs = {
            MakePath(testFolderPath, "Test"),
            MakePath(testFolderPath, "Test", "dir1"),
            MakePath(testFolderPath, "Test", "dir1", "dir2"),
            MakePath(testFolderPath, "Test", "dir1", "dir2", "dir3") };

        foreach (string dir in testDirs)
        {
            expectedDirs.Add(dir);
            Directory.CreateDirectory(dir);
        }
    }
}
```



```
        expectedDirs.Sort();

        string[] testFiles = {
            MakePath(testFolderPath, "Test", "dir1", "file1.txt"),
            MakePath(testFolderPath, "Test", "dir1", "file2.txt"),
            MakePath(testFolderPath, "Test", "dir1", "dir2", "file3.txt"),
            MakePath(testFolderPath, "Test", "dir1", "dir2", "file4.txt") };

        foreach (string file in testFiles)
        {
            expectedFiles.Add(file);
            FileStream str = File.Create(file);
            str.Close();
        }
        expectedFiles.Sort();
    }

    [Test]
    public void TraverseDirectory()
    {
        string testPath = Path.Combine(testFolderPath, "Test");

        DirRecorder dirRecorder = new DirRecorder();
        DirectoryTraverser trav = new DirectoryTraverser(dirRecorder);
        trav.Traverse(testPath);

        Assert.AreEqual(expectedDirs, dirRecorder.Dirs);
        Assert.AreEqual(expectedFiles, dirRecorder.Files);
    }

    [TearDown]
    public void TearDown()
    {
        Directory.Delete(testFolderPath + "Test", true);
    }
}
```

This completes the implementation of the integration test for `DirectoryTraverser`. Running the test with the NUnit console gives the following output:

```
NUnit version 2.4.3
Copyright (C) 2002-2007 Charlie Poole.
Copyright (C) 2002-2004 James W. Newkirk, Michael C. Two, Alexei A. Vorontsov.
Copyright (C) 2000-2002 Philip Craig.
All Rights Reserved.

Runtime Environment -
  OS Version: Microsoft Windows NT 5.1.2600 Service Pack 2
  CLR Version: 2.0.50727.832 ( Net 2.0.50727.832 )

.
Tests run: 1, Failures: 0, Not run: 0, Time: 0.188 seconds
```

The NUnit GUI gives the satisfying green bar. I successfully ran this test on both Windows XP and SuSE [SuSE] Linux under Mono [Mono].

## Acknowledgments

Thank you to Kevlin Henney for guidance and sanity checking and the members of accu-general for healthy discussion on testing techniques. Thank you to Caroline Hargreaves, Roger Orr and Adrian Fagg for review.



## References

[VFDC#] Visiting Files and Directories in C#.

<http://www.marauder-consulting.co.uk/articles.php>

[EnumMethod] <http://www.two->

[sdg.demon.co.uk/curbralan/papers/ATaleOfThreePatterns.pdf](http://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

[Fake]

<http://martinfowler.com/articles/mocksArentStubs.html#TheDifferenceBetweenMocksAndStubs>

[CI] Continuous Integration: [http://en.wikipedia.org/wiki/Continuous\\_Integration](http://en.wikipedia.org/wiki/Continuous_Integration)

[NUnit] <http://www.nunit.org/>

[SuSE] <http://www.novell.com/linux/>

[Mono] <http://www.mono-project.com/>