Two good news: file I/O is unit-testable, and it is surprisingly easy to do. Let's see how it works!

A software no-one asked for

First, we need a piece of software that deals with files and that has to be unit-tested. The TestableIO project does the following:

- given a directory, enumerate all the lossless audio files (e.g. FLAC ones) and all the lossy audio files (e.g. MP3 ones),
- if an audio file the same name is present in both lossless and lossy format, delete the lossy file
- repeat for the subfolders of the given directory

As you can see, this is just a toy software to showcase the testing part. The class DuplicateAudioFileDeleter implements the requested behaviour:

```
public class DuplicateAudioFileDeleter {
    private readonly IFileSystem fileSystem;
    private readonly HashSet<string> lossyFileExtension = new HashSet<string>() {
        ".MP3",
        ".MP4",
        ".AAC",
        ".MPC"
    };
    private readonly
```

```
HashSet
<
string
> losslessFileExtension
=
new
HashSet
<
string
>() {
".FLAC"
,
".APE"
,
".WAV"
};
public
DuplicateAudioFileDeleter(
IFileSystem
fileSystem)     {
this
.
fileSystem
=
fileSystem;     }
public
DuplicateAudioFileDeleter() :
this
(
new
FileSystem
())     {               }
///
<
summary
>

///
check the given path for files with the same name but different formats, and deletes lossy files if a lossless one exists

///
</
summary
>
```

```
///
<
param
name
=
"
path
"
>
the top level directory to start searching
</
param
>

///
<
remarks
>
this method will search in all subfolders of the given directory
</
remarks
>

public
void
CleanupDirectory(
string
path)     {
HashSet
<
string
> losslessFiles
=
new
HashSet
<
string
>();
// stores full file name without extension of lossless files

List
<
string
> lossyFiles
=
new
```

```
List
<
string
>();
// stores full file name of lossy files

// get all files from the given path and all subfolders

var
allFiles
=
fileSystem
.
Directory
.
GetFiles(path,
"*.*"
,
SearchOption
.
AllDirectories);
// build list of lossy and lossless files

foreach
(
var
currentFile
in
allFiles)        {
var
currentFileExtension
=
fileSystem
.
Path
.
GetExtension(currentFile)
.
ToUpper();
if
(lossyFileExtension
.
Contains(currentFileExtension))            {
// lossy file found
                lossyFiles
.
```

```
Add(currentFile);                               }
else
if
(losslessFileExtension
.
Contains(currentFileExtension))                 {
// lossless file found

var
currentFileWithoutExtension
=
fileSystem
.
Path
.
Combine(fileSystem
.
Path
.
GetDirectoryName(currentFile),                  fileSystem
.
Path
.
GetFileNameWithoutExtension(currentFile));                              losslessFiles
.
Add(currentFileWithoutExtension);               }
// not an audio file
        }
// deleted lossy files if a lossless file with the same name exists

foreach
(
var
currentLossyFile
in
lossyFiles)         {
var
currentLossyFileWithoutExtension
=
fileSystem
.
Path
.
Combine(fileSystem
.
Path
```

```
.
GetDirectoryName(currentLossyFile),                fileSystem
.
Path
.
GetFileNameWithoutExtension(currentLossyFile));
if
(losslessFiles
.
Contains(currentLossyFileWithoutExtension))            {
// duplicate file found
                fileSystem
.
File
.
Delete(currentLossyFile);                              }        }     }        }
```

The key point is using an IFileSystem instance instead of the usual System.IO classes, after adding to the project the [System.IO.Abstraction NuGet package] by Tatham Oddie, so instead of calling File.Delete, we use IFileSystem.File.Delete.

The algorithm detailed above is recursive, as the code should parse the root folder, eliminating duplicate files, and then repeat the procedure for all the subfolders, and so on. But as Directory.GetFiles can recursively enumerate all files in subdirectories, let's take a shortcut and get the whole set of files in a single call:

```
var allFiles = fileSystem.Directory.GetFiles(path, "*.*", SearchOption.AllDirectories);
```

Then we build two lists of audio files, one list of lossy files, and another or lossless files, and finally we iterate the list of lossy files searching for a lossless audio file with the same name, and if found, we delete the lossy file.

Done! So let's go testing.

Does it work?

There are a few scenarios that have to be tested to have to be tested. But first, the plumbing: we create a unit-testing project that MSTest, and we add the following two NuGet packages:

- System.IO.Abstractions
- System.IO.Abstractions.TestingHelpers

Now we are ready for the first test, checking if the method under test really does delete a lossy audio file in the same folder of a lossless one:

```
 [TestMethod]  public void TestDeleteOfSingleLossyFile() {      var lossyFileName = Path.Com
bine(testPath,
"myfile.mp3"
);
var
losslessFileName
=
Path
.
Combine(testPath,
"myfile.flac"
);
var
fileSystem
=
new
MockFileSystem
(
new
Dictionary
<
string
,
MockFileData
>     {          { lossyFileName,
new
MockFileData
(
"Lossy file"
) },          { losslessFileName,
new
MockFileData
(
"Lossless file"
```

```
) }                    });
Assert
.
IsTrue(fileSystem
.
FileExists(losslessFileName));
Assert
.
IsTrue(fileSystem
.
FileExists(lossyFileName));
var
audioFileDeleter
=
new
DuplicateAudioFileDeleter
(fileSystem);      audioFileDeleter
.
CleanupDirectory(testPath);
Assert
.
IsTrue(fileSystem
.
FileExists(losslessFileName));
Assert
.
IsFalse(fileSystem
.
FileExists(lossyFileName));  }
```

The setup phase holds the secret for properly using the TestingHelpers: instead of instancing a FileSystem, we create a MockFileSystem and, inside the mocked file system, two files with the proper file names. This mock of the file system is then passed to the DuplicateAudioFileDeleter so that actions on the file system, instead of happening on the real file system, are directed to the fake one. After calling CleanupDirectory, we check that the lossy audio file was really deleted.

Next tests: check if multiple lossy files, or multiple lossless files, are handled correctly. Nothing of interest here, as it is just one more file in the mocked file system. Ditto for checking that non-audio files are not deleted. So let's skip to the last test: check if a lossy file in a subdirectory with the same name of a lossless file is not deleted, as they belong to different folders:

```
 [TestMethod]  public void TestNoDeleteOfFileInDifferentPath()  {      var lossyFileName = Path
.
```

```
Combine(testPath,
@"lossymyfile.mp3"
);
var
losslessFileName
=
Path
.
Combine(testPath,
"myfile.flac"
);
var
fileSystem
=
new
MockFileSystem
(
new
Dictionary
<
string
,
MockFileData
>     {          { lossyFileName,
new
MockFileData
(
"Lossy file"
) },          { losslessFileName,
new
MockFileData
(
"Lossless file"
) }      });
Assert
.
IsTrue(fileSystem
.
FileExists(losslessFileName));
Assert
.
IsTrue(fileSystem
.
FileExists(lossyFileName));
var
audioFileDeleter
```

```
=
new
DuplicateAudioFileDeleter
(fileSystem);      audioFileDeleter
.
CleanupDirectory(testPath);
Assert
.
IsTrue(fileSystem
.
FileExists(losslessFileName));
Assert
.
IsTrue(fileSystem
.
FileExists(lossyFileName));  }
```

It could not be easier, just specifying the full path of the lossy file does the job!

For the full set of tests, please refer to the source files in my repository in GitHub .

Bonus: does it run fast?

Finally, now that we know that it does run properly, let's take a look if it can run faster. The code that extracts a full file name without file extension is suspicious:

```
  var currentFileWithoutExtension = fileSystem.Path.Combine(fileSystem.Path.GetDirectoryNa
me(currentFile),      fileSystem
.
Path
.
GetFileNameWithoutExtension(currentFile));
```

All this just for cropping the file extension? Let's test alternative by creating a benchmarking project that uses BenchmarkDotNet :

```
  public class FileExtensionRemover  {      private static readonly string testPath = @"c:foldersu
bfolderfile.txt"
;
public
```

```
FileExtensionRemover()  {  }  [
Benchmark
]
public
string
UsingPathMethods()  {
return
Path
.
Combine(
Path
.
GetDirectoryName(testPath),
Path
.
GetFileNameWithoutExtension(testPath));  }  [
Benchmark
]
public
string
UsingStringManipulation()  {
string
fileExtension
=
Path
.
GetExtension(testPath);
return
testPath
.
Substring(
0
, testPath
.
Length
-
fileExtension
.
Length);  } }
class
Program
 {
private
static
readonly
string
```

```
testPath
=
@"c:foldersubfolderfile.txt"
;
static
void
Main(
string
[] args)      {
// test both methods

FileExtensionRemover
remover
=
new
FileExtensionRemover
();
Console
.
WriteLine(
"Using path methods: "
+
remover
.
UsingPathMethods());
Console
.
WriteLine(
"Using string manipulation: "
+
remover
.
UsingStringManipulation());
// benchmark

var
summary
=
BenchmarkRunner
.
Run<
FileExtensionRemover
>();      }  }
```

On our left, inside the UsingPathMethods() method (with the critical [Benchmark] attribute), the code fragment that is used in the project:

```
 [Benchmark] public string UsingPathMethods() {     return Path.Combine(Path.GetDirectory
Name(testPath),
Path
.
GetFileNameWithoutExtension(testPath)); }
```

On our right, inside the UsingStringManipulation() method, our challenger:

```
 [Benchmark] public string UsingStringManipulation() {     string fileExtension = Path.GetExte
nsion(testPath);
return
testPath
.
Substring(
0
, testPath
.
Length
-
fileExtension
.
Length); }
```

All we have to do now is do a release build and let it run to discover the new champion:

Method	Mean	Error	StdDev
UsingPathMethods	2,456.8 ns	23.065 ns	20.447 ns
UsingStringManipulation	371.5 ns	2.912 ns	2.724 ns

The challenger is definitely faster! And on that bombshell, it's time to end this article.