Cross Platform Duplicate File Finder

Introduction

We have all been in the situation when we see our hard disk filling up and we don't know why. We are sure we can save some space, but not sure exactly where to start. A common scenario is duplicate files. Yes, we copied those pictures from mom, but our brother also gave us some pictures on his thumb drive and you know what, mom had given him the same pictures she gave us. Now we have multiple copies of our daughter’s video shot in 1080p, and while they are really cute, we do want to save some space! Enter the Duplicate File Finder!!

Design Goals

From the outset I had a few design goals in mind. I am listing these according to priority:

* Efficiency  
    
  - Minimize the number of files that needed to be compared.  
  - Minimize the number of times a file needs to be scanned  
  - Keep the algorithm as simple as possible

* Portability  
    
  - Can run as a console App on Windows or the Mac  
  - XAML interface available to be used either as a windows store app, desktop app or mac desktop app
* Iterative Build  
    
  - Get something done and working ASAP and keeping adding to it as time goes on  
  - Functional from day 1 and after every release build

Algorithm Overview

I came up with the following algorithm.

1. Group files by file size. All files having the same size would be grouped into the same bucket. Any bucket with two or more files would be considered possible duplicates.

class FileMap<T> : Dictionary<T, List<IFile>> { }  
class FileMapBySize : FileMap<long>

1. Now that the list of possible duplicates had been narrowed down to files with the same size, I created MD5 hashes and grouped the files by MD5 hash. Any group in which two or more files shared the same MD5 was a confirmed duplicate.

class FileMapByString : FileMap<string> { }

Abstracting the File System

Since I wanted to maintain code portability, I had to abstract away the File System.

Note: Since the Win RT runtime does not support synchronous file system access, this interfaces needs to be asynchronous. Note: I managed to keep the GetSize method synchronous by performing async size retrieval during object construction (see next page) I elected not to do this for GetUniqueHash as (obviously) we do not want to calculate the hash code for every file we scan (a tremendous waste of time) The console version of the application app uses a 'fake asynchronous' interface since a synchronous interface can pretend to be asynchronous and just return immediately, but you cannot go vice-versa.

public interface IFile : IFileSystemEntity

{

long GetSize();

Task<string> GetUniqueHash();

Task Delete();

Task Write(StringBuilder b);

bool IsHashed();

}

public interface IDirectory : IFileSystemEntity

{

Task<IEnumerable<IFile>> GetFiles();

Task<IEnumerable<IDirectory>> GetSubDirectories();

}

Abstract File System Implementation

Two different implementations of the file-system interface can be seen below, one for the console interface and one for the WinRT runtime

WinRT: Use a static “constructor” so that the async get file-size operation can be performed as part of “object construction” Use StorageFile object for file access as mandated by WinRT

public async static Task<File> GetNew(StorageFile file)

{

File f = new File(file);

await f.CalculateFileSize();

return f;

}

private async Task CalculateFileSize()

{

var p = await m\_file.GetBasicPropertiesAsync();

m\_size = (long) p.Size;

}

public long GetSize()

{

return m\_size;

}

Console App implementation is much simpler. Uses simple string paths for file access and synchronous System.IO function calls.

public File(string path)

{

\_path = path;

m\_size = (new FileInfo(Path)).Length;

}

public long GetSize()

{

return m\_size;

}

Using Events to communicate with the UI

The UI (both console and XAML) are updated via events. Console implementation is simple and the message is just written to the console. In the XAML code, the event updates the View Model ( an observable collection ) and this in turn fires its own events to update the UI.

public delegate void DuplicateFound(string hashcode, string filepath);

public event DuplicateFound OnDuplicateFound;

Console:

private static void dff\_OnDuplicateFound(string hashcode, string filepath)

{

Console.WriteLine("Found Dup:" + filepath);

}

XAML:

void dff\_OnDuplicateFound(string hash, string path)

{

var viewmodel = this.DataContext as DuplicatesViewModel;

viewmodel.Data.Add(new DuplicateGroup(hash, path));

}

See the complete code for more details …