COSC346 Assignment 1 - Media Manager Library

Team SMAX:

Sam Paterson (3175949)

Max Huang (4427762)

Project Specifications

To design, implement, test, and document a tool for managing a media library - building upon nothing more than the Foundation framework.

The tool must be able handle a large collection of media of assorted types (images, video, music, text documents) as a library and manage it - including a set of metadata attached to each of these types of media.

The library must support the following features:

- Import and export of data from files.
- Searching metadata for keywords - and displaying a list of the files returned by the search.
- Adding, changing/setting, and removing metadata keywords or values.

Object Oriented Concepts in Design and Implementation

When planning and designing our program at the start, we wanted to decouple the front and back end as much as possible. This would make it easier and more efficient to work on as a pair while simultaneously allowing changes to be made to the front end without affecting the back end, and vice versa. The MM_collection class uses protocols for the functions, and thus the front end can be confident that any calls to the back end will always be implemented.

Another design principle we adopted was inheritance - when creating the command classes that the main class calls, we quickly realised that there was significant amount of repeated code in the variables and also in the `init()` functions. As such, we decided to use inheritance to reduce the amount of duplicate code by initialising and storing most the variables in the parent class `CommandParent` and using the parent's `init()` function rather than repeating the same `init()` function for each class. We also created a function in the parent class that the children can use to help them make temporary files and metadata since we have multiple children wanting to use this function so it made sense to create it in the parent class.

Code testing

We implemented two types of testing: unit tests and bash script testing. We used unit testing in a file called `collectionTesting.swift` to test the individual back end components (e.g. testing `add(file)` and `add(metadata, file)` functions). Testing in this manner allowed us to be confident that the back end is working as expected. It also meant that if there were any issues with the program as a whole, we know it is most likely in the front end. The only issue we encountered with unit testing was that some of a functions, specifically remove functions, required users to validate that the action taken is desired via `stdin` (using `y` or `yes`), we couldn't mock `stdin` for the unit tests so we had to test some of these as part of bash script testing, which is less than ideal, especially since it should be part of the unit tests.

We used bash script testing to test the program as a whole to make sure the user experience is pleasant, and the program does what is required and expected. We are testing with a large range of inputs and commands in various orders so that we could deal with anything that the user could attempt. Due to unit testing for the back end, we know if there are any problems it is an issue with the front end which made debugging much easier. Bash script testing also allows us to test for edge cases and invalid inputs - being a manual user testing scheme, it is unlikely that we were able to think of all possible test cases, but we should have covered most cases.

Based on our unit testing and bash script testing decision, we have not incorporated a testing functionality within the `main.swift` file. However, there is a line of code `if commandString.first == "#" { print(""); continue }` that is used to ignore our comments that we put into our bash testing script. The comments were used to separate out the different testing cases and for better readability.

Role separation as a pair

To decide the roles, we both read the specifications and Sam had an idea on how to do the back end (collection class functions) so he started working on that while Max had an idea about working with JSON files, so he started working on the front end stuff. We then worked together to integrate our code. Sam then did the unit tests because he mainly worked on the back end while Max did bash script testing because he worked mainly on the front end.

This method of role separation worked well because we were able to work on different files and aspects, and thus we had hardly any conflicts when merging code. In terms of getting in each other's way or slowing the other down we never had any problems because Sam started and finished the back end almost before Max started so Max was able to use his code straight away thus streamlining production. Separating the testing also worked well because we were testing our individual parts so were able to fix any issues quickly because we each wrote the code we were testing and thus could easily locate the problem.

Since there was clear role separation, if there was an issue, the person responsible for that part of the code would go and fix it, but we both have an overview of how each other's code works and thus a holistic view of our program.

Additional functionality added

We believe we should be awarded 3 bonus marks because we added at least 3 elements of extra functionality.

The first being that we check to make sure that when a JSON file is imported, we don't add any duplicate files to the collection by checking the fullpath of the file, as contained in the JSON. This was a design decision because our system works fine if they do import a duplicate JSON file, but we couldn't think of a circumstance when a user would ever want to do this, so if this situation did occur it would be a mistake and could confuse the user leading to an unpleasant experience. We want the user to have a streamlined experience, so we have disabled this functionality so users get notified when trying to import a JSON file that has already been imported.

The second thing we added is a detailed list which shows the filename, type, and metadata for a specified file(s) by using an index after using a `list` or `search` command. We choose to do this because we often wondered if the changes we made to a file (e.g. adding and deleting) did what was intended and figured the user might wonder this as well. This functionality lets a user see a detailed view of the file without having to export the file to a JSON in order to see changes or try search for the key/value and make assumptions that it has done what was expected.

We added a remove all function so that you could clear the whole collection easily without having to restart the program, this could be useful if an error was made in all the imported data or if the user just wants to remove all files.

The last functionality we added is a user confirmation prompt when deleting an item from the collection. This is a simple addition but an important one nonetheless, as it can save the user a lot of heart ache if they accidentally type the wrong key or command etc. By default the response is set to `no`.

Assumptions made

- If duplicate keys exist in a file when importing, the first occurrence of the key is stored and ignores any further metadata with the same key.
- When setting a new value for a duplicate key (for a file), there is no issue as to which one of the duplicate keys should be changed because the program works by first removing all instances of that key and then adding a new key/value pair with the original key and new value as the new key/value pair. This results in removing all duplicates for that key.
- Exported metadata as JSON does not appear in the same order as when they were imported.
- When searching for multiple keywords, if a single file matches multiple times only one instance of that file is displayed as the result to the user.