Vivian Breda 1663266
Nikolah Pearce 6377167

346 Assignment 1

Object-oriented concepts

The protocols in the foundation framework we were given were very difficult to work with. Methods were ambiguous and a little unclear, most likely because the comments were as well. There were multiple areas where the protocols restricted our implementation and design e.g. due to not being able to add to or modify the provided protocols. We wanted to take an approach that followed programming to an interface e.g. `var files: MMCollection = Library()`. Declaring our variables as their protocol types, but instantiating these as their concrete classes (that extend those protocols). However, this was not always possible. We added many extensions to the original protocols but some variables could not be added as extensions - Xcode simply would not allow it. This limited our functionality of some variables e.g. a property `count` defined in `Library`, was inaccessible via `files.count` as 'type MMCollection has no member named count', and adding the property as an extension required a concrete implementation of a computed variable - which couldn't access the number of files and was as such, useless . Currently, as the program stands, there are some areas that program to the interface as we desired, and some areas that simply could not - so were both declared and instantiated purely by their concrete class. If the extension for `MMCollection` had worked as we wanted, we would have added all of the functions we added in to be part of the extension, including: `removeAllFiles`, `remove`, `loadDictionaries`, `rmvDictionaries`, `updateDictionaries`, `listByType`, `isDuplicate` and `isMetadataDuplicate`.

`FileValidator` was made to separately handle the validation and checking of `MMFiles` and `MMMetada`. This class has four static dictionaries that store valid metadata, and any type validating refers to these original declarations. This provides a means to easily update definitions of types and their required metadata, without affecting functionality of the program as a whole. We designed the `FileValidator` class this way to have as loose coupling as possible with the `File` and `Metadata` classes. File needs to know absolutely nothing about the validation, and everything that `FileValidator` needs to know about the file is stored in static dictionaries/data fields. We also demonstrate high cohesion by the validation class specifically having only validation methods. Each relate to the one unit goal of converting a `Media` struct (from the JSON decoder) into a validated `File`. This also tied in nicely with keeping abstraction a principle of our project - the user does not need to know anything about the internals of the validation class, except that when they call the `validate()` method, everything gets done perfectly.

One downside of the protocol issues above is that our computed variable `type` in the `MMFile` extension is one of the most poorly designed OOP methods found in the entire project. It currently is entirely static and independent of the `FileValidator` class, but encompasses the principles of the class as hardcoded if/else statements. Unfortunately, we couldn't figure out how to improve it. We originally wanted to have a stored property `type` in the concrete `File` class that would be initialised and set when a `File` was created - which worked fantastic, except was never usable. By this point we had designed all four `File`

instances to be programmed to the `MMFile` interface and we once again fell into a problem with not being able to edit the protocols nor use an uncomputed variable in an extension of the protocol. The computed variable `type` we have currently, is the best solution to the problem we found - though it would be the first thing we would work on improving in futurel

Other OOP principles we have included in our project include encapsulation, inheritance and ensuring our code doesn't 'smell' where possible. We encapsulated data fields across our files, creating private internals and public getters and setters (often within Swift computed variables) for them. We respected encapsulation as best we could. We used both inheritance and composition when modeling the media files and their metadata. Inheritance for the individual file types e.g. document vs image, all extending the generic `File` class. Composition was already defined in the protocols as `MMFile` types have a `MMMetadata` array. We also tried to keep our methods distinct and take as minimal parameters as possible, minimising the chance of 'code smell'. For example, the `FileValidation` class takes only one single media as a parameter, but performs all functions needed for validating.

The only additions or corrections we made to the original framework provided were minor, such as adding a `getAll()` method into the `MMResultSet` protocol (that returned all files in the result set). We heavily utilised the `MMError` enum, adding many more cases of our own choosing to elegantly handle errors as we saw fit. All error throwing is successfully caught and dealt with in a single class, `main.swift`, where the information is printed to the user via the console. The `commands.swift` file occasionally does print to the console too, which we questioned as to whether it is the best procedure. However, we ultimately weighed usability and clarity (of the tasks being performed and their success or failure) higher than keeping all print statements in our `main.swift` file. For example, when the load command is called, the user is informed of successful loads, unsuccessful loads, and any duplicate media found.

Testing

In the early stages of the assignment we started testing manually and visually checking things were as expected. This was supplemented by printing out not just the filename of the media in our library, but printing out all the associated metadata as well. Testing things like "add" and "set" were very easy to confirm they worked because we could scan the list of metadata for the updated keypair. One of the last adjustments we made before submitting was to hide the metadata from the print list, as indicated in the assignment example statements.

For our testing framework, we started to implement basic Swift Unit Tests, following the XCTest examples. However, we ran into many problems with setting these up including creating the bundle, connecting the media library classes, calling XCTest asserts, and even initialising data fields via the `setUp()` method. Eventually we gave up on the official unit tests, and decided to test our own. We used tick emojis to represent implemented tests, and cross emojis to note the yet unimplemented ones. We created our own `setUp()` and `tearDown()` methods that would return our data fields to original states before each test was run.

We used the framework to test all major functions used in our library, including all `Library` functions, all `Commands`, and the `FileImporter` and `FileExporter` classes, as well as basic tests for creating and storing `File` and `Metadata` instances. We also implemented the testing framework so that it can be called from a standard start and build of the project - through the user prompt. Entering the keyword “test” will trigger all tests to run. This was also added to the “help” menu.

Teamwork

Nikolah implemented the testing framework, file importing, file validation and error throwing/handling. Vivian implemented file exporting, command handling, inheritance through the files and library functions. There was fantastic communication between both of us and all design choices were discussed and agreed upon before implementing. Any bugs were also discussed as a team and fixed and tested together before proceeding. We used an issue driven development method to focus on tasks, and the commit history on GitHub should show an equal amount of work by both parties.

Extensions

Regarding invalid files or metadata, we chose instead of error handling during validation via a catch and throw, we would gather errors as `String` and relay them to the console at the end. This allowed us to mimic the functionality of a real media library, where loading many files all at once, say 1000, and 1 was misplaced or had an error, would not stop the other 999 files from loading successfully - as would be the case in a real life library. It simply displays which files (if any) weren’t validated, the total count, and why. It also displays which ones validated successfully and the total count of those. This is extension number one that we implemented.

Duplicate media was excluded from being added. We classed duplicate media the same as any standard computer system, that being a file with an identical path and filename. Metadata was not considered in the duplicate check. Similar to above, instead of throwing when duplicate media is detected, we simply gather the errors as a `String` and display them at the end, alongside any media added that was not duplicate. This is extension number two.

We added extra functionality to the `ListCommand` - listing files by types. If you call ‘list image’ or list ‘video’, only those files of that specific type will be displayed. Again, this was inspired by the functionality of a real media library where you can view and sort by types. This is extension number three.

During the exportation/save process, we also added functionality so that the user can choose where they want their file to be saved, either in their Home directory (~/etc), the programs working directory, or a specific directory as provided by the fullpath. When a user provides a filename, the program detects which of the three conditions/directories it is being asked to save to and continues from there. This is extension number four.

In order to preserve the exported state of the media library data, we also ensure that each file saved has the extension `.json`. During testing we noted that if the user entered a

filename and an extension as input, the program would attempt to save the json data as that extension. This was an awful flaw and security risk. We implemented a very simple basic fix by appending the extension `.json` to every filename provided - regardless of whether it had an extension already of not. We now have a guarantee that our Media Manager Library can only save files in the expected `.json` format. This is extension number five.

Because of the fantastic features implemented above we believe we deserve all three bonus marks for extensions. Not only that, but our testing framework exhausted practically all of our methods across all of our classes. The sheer effort put into efficiently testing like this shows how clearly we considered this assignment and that we deserve the extension bonus marks.

Reflections

While performing our final tests, we realised that there were still a few issues left that we would have liked to work on, provided we had the time for it - because we didn't have previous experience coding in Swift, we spent a large portion of this assignment reading through the Swift Documentations and other specialized tutorials. The things we noticed that could be improved include:

- Warning before overwriting `.json` files with save/save-search commands;
- Forcing exported files into `.json` format more elegantly - currently used a very basic fix;
- Prompting to save before quitting if the last search hasn't already been;
- Implement true Swift unit testing.