Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Automated metadata tags management tool for MP3 files

Estudante: Juan Ignacio Muñiz Gómez

Dirección: David Cabrero Souto

A Coruña, September de 2023.

Abstract

Managing a collection of files can become a tiresome process when said collection grows large enough. This problem becomes quite apparent with music collections, since it's fairly common to find issues of consistency in the nomeclature and formatting of products provided by different services. Thus, the aim of this project was to create an open source desktop graphical application that could automate the process of maintenance, given a user's input. The project was developed in python, using the wxpython library for the graphical elements, and music_tag for the metadata editing.

Resumo

Xestionar unha colección de ficheiros pode volverse tedioso cando a colección crece o suficiente. Este problema faise moi aparente coas coleccións musicais, xa que é bastante común atopar problemas de consistencia na nomeclatura e no formato dos produtos que ofrecen distintos servizos. Polo tanto, o obxectivo deste proxecto era crear unha aplicación de escritorio gráfica de código aberto que puidese automatizar o proceso de mantemento, dada unha entrada do usuario. O proxecto desenvolveuse en python, utilizando a biblioteca wxpython para elementos gráficos e música_tag para a edición de metadatos.

Keywords:

File management

Task automation

Audio files

Open source

Palabras chave:

Xestión de arquivos

Automatización de tareas

Arquivos de audio

Código aberto

Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Motivation

Managing a collection of files can become a tiresome process when said collection grows large enough. This problem becomes quite apparent with music collections, since it's fairly common to find issues of consistency in the nomeclature and formatting of products provided by different platforms, but there is no easy solution for the average user to work with large batches of files.

These inconsistencies are simple enough to fix when they only affect the name of the file, but more often than not, the real problem stems from the metadata tags. Metadata tags are additional fields of information encoded within an audio file that hold information about said file. This information can be technical, such as the audio refresh rate, but more importantly to our ends, it can contain information about the production of the song such as the name of the artist, name of the song, name of the album, and even an image to depict the cover art of the album. These fields are more relevant to the end user because they're commonly used by music players instead of the filename when they're available, so if your music collection has a mix of properly tagged songs, songs from different sources with different formatting for the tags, and song without any tags at all, it can create an environment that is needlessly difficult to navigate.

The aim of this TFG was to develop a desktop application with a straight forward and legible interface that would assist users in editing the metadata tags of their audio files.

1.2 Objectives

There are certain requirements the end result must meet for the project to be considered successful:

- Metadata management: The application must be able to read batches of audio files and

allow the user to modify their metadata.

- User readability: The application must provide an interface capable of automating user actions without the need for prior coding knowledge or any extensive training to use the application.
- Expandability: Rather than trying to adapt the project to every possible edge case that could arise, it is more prudent to make it future proof by creating an open source platform, so that any user can adapt it to their specific needs or embed it with new technologies.

Chapter 2

Discussing the technology of the project

2.1 Already existing tools

This project began not because of a lack of tools to deal with the issue of audio file metadata editing, but because all tools available online suffered from one of two major drawbacks. These can be easily illustrated by explaining the features of two popular programs that are used for metadata editing:

- **tagmp3.online**[1] is a handy browser-based solution that allows you to upload any audio file to edit the filename, relevant metadata tags, and even add album art. It is extremely convenient and straight-forward, but it offers no extra functionality: all information must be provided manually, and only one file can be edited at a time.

  We arrive then at the first issue: **lack of automation**. As useful as this website might be, it's of no use when trying to deal with a collection, unless the user is willing to invest a sizeable amount of time.

- **mp3tag**[2] is a desktop application chock-full of features. It can distribute information from one tag or filename to another, it has a plethora of conditional functions to cover all sort of user-defined situations, it has regex support, it can deal with large batches of files while still having a simplified interface for manual file editing... However it, along with many other desktop solutions, works on a code-based pattern matching basis, leaving it in an awkward spot where basic users will have a difficult time understanding the program, while still requiring advanced users to learn the complexities of its grammar to perform any action separated from the base uses. And speaking of base uses, the complex infraestructure often means that open source developers will more than likely

have a hard time embedding any new features they may like, as new technologies and needs arise for the user.

We can refer to this second issue as a problem of **complexity**, both in user readability and openness to further development.

From this short analysis, it's clear to see that the objectives set forth in section 1.2 were a direct effort to overcome these two liabilities. We want an application with powerful features, but takes little to no training to use, and that can be expanded upon to allow embedding with other technologies.

2.2 Technology employed

This project is a continuation of a command linfe utility that was originally developed in python, and because of python's popularity, it seemed like an appropriate tool to continue the project, both because of the variety of libraries its popularity abides, and because it meant that any developer who wants to work with the source code of the tool has a good chance of being familiar with the language already.

Two libraries were used in the development of the project:

- **Wxpython**[3], for building the graphical elements of the application. Wxpython was an attractive choice for the GUI framework because it uses native widgets from the current operating system, which helps with the readability of the program, but will also assist with portability.

- **Music_tag**[4], for interacting with audio file metadata. There are plenty of metadata editing libraries, but many of them require the use of separate modules to interact with different file extension. Music_tag does offer this layer of abstraction, which simplifies development.

  With that said, the original draft for the project only took MP3 files into consideration because of their popularity among end users. However, to test the flexibility of the library, extensive testing was also done with WAV and FLAC files. These two file extensions were chosen on the basis that they are lossless and thus more commonly used in production environments, which we wanted to support as well.

Chapter 3

Details of development

3.1 Development methodology

Because this was the author's first work with graphical interfaces, an agile iterative methodology based on prototypes seemed prudent in order to ensure steady and measurable progress. The final product was divided into four stages of usability, which would each have a stage of analysis, design, implementation, and testing. The defined stages are as follows:

- 1) Acquiring an understanding of wxpython and building a graphical framework.
- 2) Loading different songs and their metadata into the program.
- 3) Allowing the user to manually edit the metadata.
- 4) Creating an interface that will edit the metadata of batches of songs based on instructions provided by the user.

3.2 Planning and tracking

The alloted time span for development was two weeks per prototype phase plus an initial three weeks for the initial proposition and design of the project, preliminary interface designs, research of the topic and technologies involved, and research and discussion of user behaviour to design the usability tests. At the end of the project an additional week was alloted to review and clean up the documentation. Each week the author would spend up to 30 hours in development, and the director would spend between one and four hours reviewing the progress done, depending on the nature of the work.

While there were some fluctuations with the planned phases of the project, the overall duration wasn't too far off from the initial estimation. Most of the irregularities came from

prototype three taking significantly less time than other phases (only one week) and prototypes one and four taking longer than initially estimated to complete due to complications with the graphical library employed.

Here's a breakdown of the hours spent on each phase:

Phase of the project	Developer hours	Director hours
Initial planning	86	25
Prototype 1	77	5
Prototype 2	59	4
Prototype 3	36	2
Prototype 4	72	7
Wrap up	32	20
Total	362	63

Table 3.1: Breakdown of time spent on the project

If we were to assume that the main developer, who also did analyst and testing work, had an average pay of 15€/hour; and the director, who acted as a step-in project leader, would have a pay of 24€/hour, this would drive the total costs of the project to 6942€.

Chapter 4
Analysis

In this chapter, we will review all the requirements identified during the planning phase. Along with each individual analysis step for each of the prototype phases, a preliminary study had to be done to determine the scope of the project. It's particularily important to set realistic and viable goals when designing programmable interfaces, since it's easy to overengineer factors that won't be relevant to the average user and will simply bloat the syntax and flow of information, creating a confusing environment for the user.

4.1 Program functionality

First, we need to boil down the abstract definition of the program into concise and realistic cases of use to define the system requirements. We do so by trying to emulate any user behaviour that could fall within the scope of the project, and discussing whether it fits the initial description of the project or not, and how that affects the extent and reach of the program.

4.1.1 Extent of the considered cases of use

When thinking about what the end result of the project will be able to do we need to keep two factors in mind: how easy would it be for a new user to reach a desired result, and how complex would the framework be for any would-be open source developer to add new functionality to the program. While it would be a commendable exercise to stay open to any future possibility or need for the developer and the user, we must stay within reason while considering just to what extent can this program be used, and to what ends some other tools could be better suited instead.

In that regard, two features were axed during this first planning phase: collection-wide conditional checks, and complex conditional statements and logic loops. The reasoning is that, while potentially useful, they would only be used to create very complicated "one style

fits all" solutions, which is not the original intent for the program, which is to save the user the hassle of repeatedly typing already existing information.

Another feature that was discarded from the original draft of the project was the ability to download images for the cover art based on existing information about a file. It was originally included because this application was a continuation of a personal project that was a command line-based metadata tag manager that also had a crawler used to automatically download images based on the name of the artist of a song. The crawler is no longer functional, since it was hard-coded to work with the current web layout of a search engine that has since updated said layout, and at this point of the project downloading images was no longer a core objective for the project. Still, a big part of the project was allowing any developer to create their own instructions, which could involve downloading images using either an API or a crawler, and in that regard the program is still perfectly capable.

4.1.2 Requirements

- The user can load a collection of sound files of any supported format.
- The user must be presented with a number of relevant data tags for every loaded file.
- The user can modify the filename or metadata of any of the loaded files. They can also mark the file for deletion.
- The user can create instructions that can recreate any behaviour the user could make using the program.
- Said instructions will only take into consideration themselves, and each file individually.

4.2 User interface

The project will be a single window application that needs to present the user with information about loaded audio files, and instructions to be executed.

4.2.1 Programmable instructions

To make the programmable instructions readable to users without coding experience, we opted for a visual programming interface. Instead of text based instructions with complex syntax, the user will select one of the available functionalities from a drop down menu, and within a self-contained graphical object, will input all data relevant to the execution of said instruction.

Each instruction object is comprised of two parts: the toolbar, which is generic for all instructions, and the body, which will change depending on the instruction the object will

perform. As can be seen in the provided screenshot 4.1, the toolbar can perform the following actions: change the behaviour of the instruction, change the condition which the file must meet for the instruction to take effect, a complimentary text entry to define the conditional statement, and buttons to run the instruction individually or discard the instruction entirely.

As mentioned earlier, the body of the object will change depending on the instruction selected. Most forms of input will be done via text entries, or by selecting target and source metadata tags.

4.2.2 Loaded files

With every loaded file we want to both show the metadata tags that will be affected by the program, and we want it to act as an interface for the user to manually edit values. This raises the question of just how much information we want to cram into each graphical element.

In the end, we opted for a conservative approach, and hand-picked tags which are more prone to be of interest to the user.

As can be seen in the provided screenshot 4.2, we have text entries to edit the filename, and three metadata tags. We also have a drop down menu to change the export options of the changes done to the file, which are to either commit the changes, ignore them, or delete the file. There's also a text field for the cover art of the song, where the user can provide the path to an image to be used. This is because when using the program, we considered that it will be more important to the user whether the path to a new image to be used is correct, rather than double-checking that the image they are using is in fact the one they prepared beforehand. Again, it was not within the scope of the program to act as an image viewer, but to facilitate the user the automation of adding images to files based on pre-existing information, and for that purpose, the path is more relevant.

4.2.3 The window

Since virtually all information presented to the user is text-based, we chose to split the window hortizontally in order to cut off as little text as possible, and separated information about the instruction and about the audio files, putting the instruction of the upper half, and files in the lower. And since we need to present several fairly large graphical items in each half of the screen, we made each half of the screen scrollable. Each half of the screen is also fitted with a toolbar with functions specific to the region they're in.

As can be seen in the image 4.3, the instruction part of the interface has buttons to add more instructions and run all present instructions, which simply tells each individual instruction to execute itself, in order. The file collection part of the interface is basically the same, with buttons to either load files or commit the changes performed in the program to disk. Of note is that wxpython can use the native file explorer to let the user choose what files to load

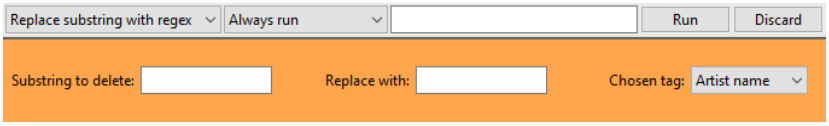

Figure 4.1: A screenshot of a programmable instruction inside the program

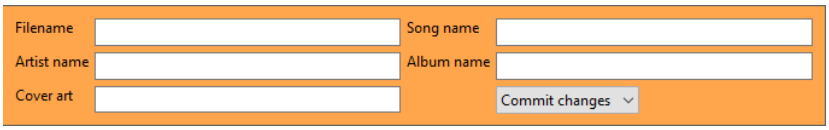

Figure 4.2: A screenshot of how the program presents all information pertaining to any single loaded file.

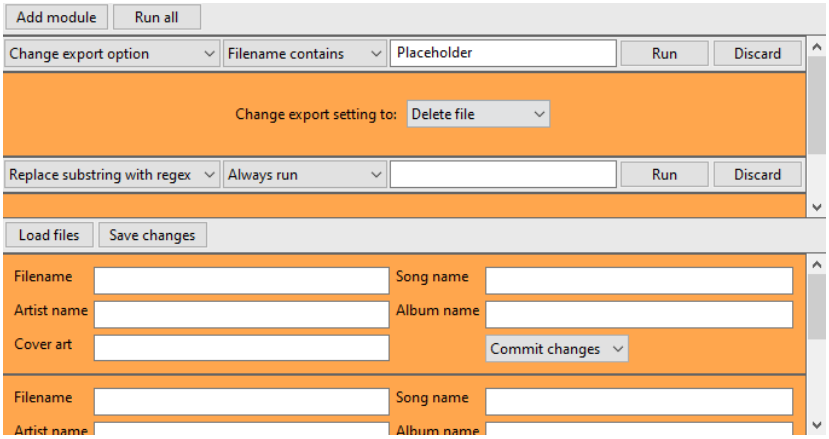

Figure 4.3: A screenshot of the graphical user interface.

within the program, which is a very natural and intuitive way for the user to interact with the program.

Chapter 5

Design

To create the visual programming interface, the command design pattern was used to create the instruction objects. The command design pattern separates the call of a method and the implementation details of said method, propagating the call from a ”command” class whose sole purpose is to act as an interfacer with the rest of the program, and the ”receiver” class, which encapsulates all information needed to perform its function.

This is a great for our project, since we need to present the user with objects that perform actions, but we can't know anything about said actions in advance. So it makes more sense to let the graphical interface act only as a platform, and then rely on the instruction object itself to perform the action by itself when called upon. This is also userful to simplify the architecture for developers to create new instructions, since we can encapsulate the logic of the instruction in a class that is separate from the main control and interface of the program

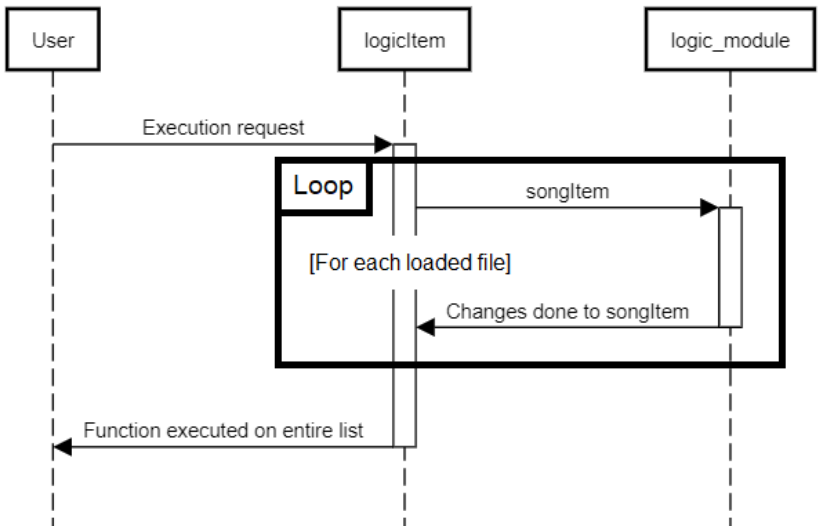

Figure 5.1: Sequence diagram to illustrate the relation between the command class and the receiver class.

We use wxpython's file explorer function to let the user select the files that will be processed by the program. This function provides a path to each file, which we then use with the music_tag library to load all relevant metadata tags about the file. For every loaded file we add a new graphical item to the interface that stores and presents to the user all relevant data found in the file. Each graphical element also has a text entry for each file the program allows to modify, so it can both present information to the user, and act as a manual editor for minor changes. All fields are stored in a dictionary that uses the readable identifier of the field as a key. These dictionaries can then be used by the instructions to dynamically show the supported fields.

Every instruction the user creates has the capacity to modify the metadata tags and filename of all loaded files, with the aim of automating data entry. The details of what the instruction does depends on user input, more commonly just redistributing or fixing already existing information. When an instruction is called to be executed, the command class of the program passes a reference to every loaded file object, and the receiver class handles the request and processes the data. All manual edits by the user, along with the modifications made by user created instructions, are performed on the data these objects hold. In doing so, we ensure that all the information needed to commit the changes to disk is encapsulated in a single object, which we then relegate the export process to, following the same principle we did with the instructions.

The export process relies entirely on the music_tag library. Its has its own method to load files given a path, and creates a mutable object, on which the tag edits are performed. For every object representing a file, we first load an instance of the file using the library method and the path stored in the object. Then we modify all the affected tags of the instance so they reflect the changes done to the object. Finally, all the information that is now stored in the mutable instance of the file is commited to disk using another method from the music_tag library.

The interface was built by creating a hierarchy of panels and sizers. This is further explained in section 6.1.

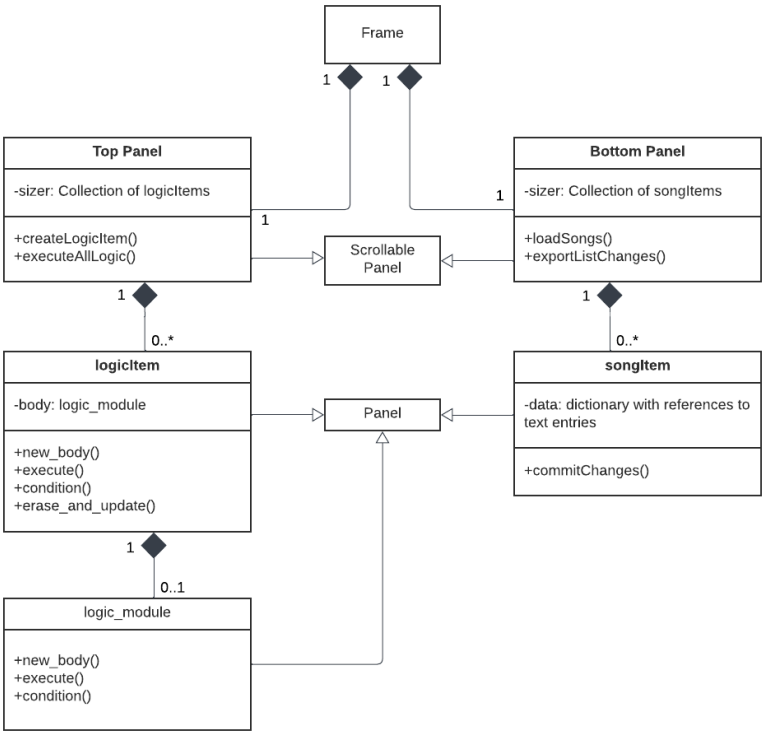

Figure 5.2: UML diagram for the architecture of the app

Chapter 6

Implementation and testing

In this section we will review the progress made with each phase of the project, explaining both the details of how each element was implemented, and the testing done trying to recreate the behaviour of a regular user.

6.1 Prototype 1: Graphical interface

Getting familiar with the wxpython library was by far the most time-consuming part of the project.

Starting from zero, wxpython first creates a window, which is referred to in code as a ”frame” class. From here on out, everything works on a hierarchy basis. We can now create widgets and items within the window, but they must have this base frame as the parent.

To simplify the decisions taken in section 4.2, first we need to split the window in two horizontal halves, and then separate each half into a toolbar of fixed size, and a scrollable body which will hold either the user created instructions, or a list of the loaded files.

Instead of manually placing every widget in our graphical interface, wxpython has a feature called ”sizers”, which evenly splits the space of the surface they are assigned into a grid. Instead of being created with a parent object, they are created individually, and then assigned to any object that has a surface. We can then append items to this sizer, and they will be placed dynamically inside the first available cell.

There’s different types of sizers with different features. These are the two most used in the project:

- FlexGridSizer divides a space into a grid of N rows and M columns (The number of rows and columns have to be specified at the time of creation). With FlexGridSizer, we can specify which rows and columns we want to expand evenly, and which should only take as much space as their contents need, which is needed to have toolbars of a fixed size.

- BoxSizer lacks many features by comparison, but has the advantage of not needing to know the number of cells it must split itself into at the moment of creation. This is useful for dynamically hosting the list of instructions and loaded songs.

To further subdivide the space inside a sizer we use a ”panel”. A panel is a relatively simple widget that draws a surface inside the parent, and allows one to add a simple border and change the color of the background. Its utility comes from the fact that it can have children of its own, much like the original frame. And just like a frame it too can be subdivided using a sizer. So if we place a panel inside the cell of a sizer, and set it so that it occupies as much space as possible within that cell, we can apply a different sizer to that panel, to subdivide already subdivided space.

Repeating this hierarchy, we split each half of the screen into a toolbar and a scrollable body, using a special kind of panel. We then place a vertical BoxSizer in each body so that we can append any number of instructions or audio files, and we put placeholder buttons in the toolbars.

We can make widgets react to user inputs using the ”bind” function. Wxpython creates events in reaction to user input, such as recognising mouse clicks, text boxes being highlighted, buttons being pushed... With the bind button, we designate a function to be called every time an event reaches any designated widget. Events are generated by the widgets that the user interacted with, and the event is then propagated upwards in the hierarchy until we reach the base frame, or one of the bound functions deals with the event and does not propagate it further. Fortunately, most of the user input is done through buttons, comboboxes, and text entries, which create very specific and manageable events, so the project didn't require any sort of complex event manager.

To test the scrollable lists, we added a placeholder function to one of the buttons that created an empty panel in one of the lists, with a button to delete itself. During testing, two problems arose because of the interface's ability to adapt to the size of the window.

Firstly, when destroying and discarding an item from a sizer, the rest of the items don't rearrange themselves if there's no call to update the layout, which normally only happens when the window is resized. So we had to add a manual layout update call every time we destroy an item inside one of the lists.

Secondly, if you shrink the window too much then some items will not be visible anymore. This is a minor problem if the items just disappear from the window because there's not enough space. However, there's also the possibility of items overlapping each other, which is not an issue at first, but if you expand the window again there will be graphical artifacts carried over in each of the affected items. An example of this issue can be seen in the rpvodided screenshot 6.3.

This can't be fixed using manual calls to update the layout, but it can be prevented using

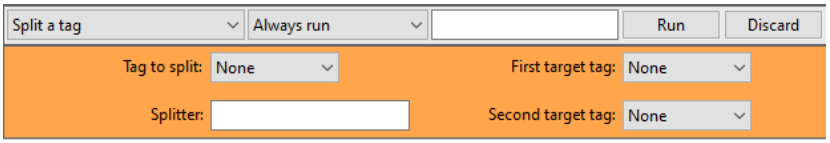

Figure 6.1: Regular screen capture of an instruction object

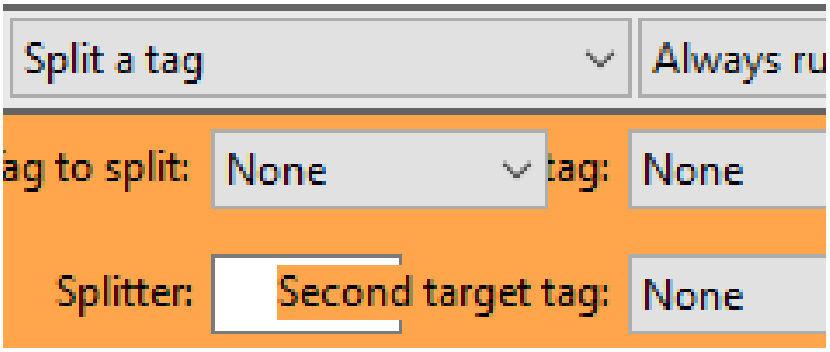

Figure 6.2: Shrinking the window so the widgets inside the instruction overlap each other

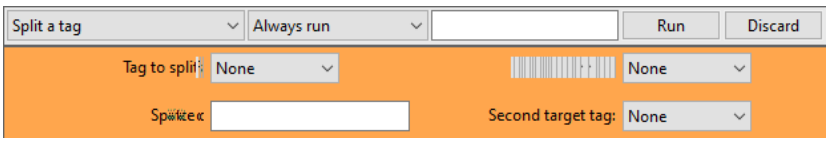

Figure 6.3: Aftermath of the interaction when re-expanding the window to an appropriate size

a property called ”minimum size”, which prevents the user from making the window smaller then the necessary space for a widget to exist. Immediately this is not a desirable solution, because at no point do we want to take away agency from the user. Furthermore, using a test item that would emulate the object that would contain the instruction, we found inconsistent and undesirable behaviours when deleting and creating objects containing children of variable minimum size. Not to mention, this would add an overhead to any further development from any third party could do with our platform, since they would have to worry about this issue too when creating new instructions.

Fortunately, the artifacts clear themselves when minimising the window, so even if the user were to find themselves in a scenario where this could be an issue, we don't need to take any extra precautions to deal with the situation. For the sake of simplifying development, we considered this a non-issue from here on out.

6.2 Prototype 2: File I/O

Development of this phase was fairly straightforward. Wxpython can use the native file explorer for the current operating system and will return the path of any files selected by the user. The user will only be able to choose files with supported file extensions, so there will be no compatibility issues on that front. The function then returns the path of every selected file.

To represent all data loaded from a file we created a class called songItem. SongItem is derived from a panel, and has the function of both storing all loaded information, and allowing the user to edit this information, which is what the panel is used for: to host all the widgets necessary to show the information gathered and allow for data entry. Details about the interface can be found in 6.1.

The program doesn't yet have the ability to read or write the metadata of the files loaded by the program; this prototype served as a middle step to ensure file I/O worked properly, and that complex panels were appended and represented properly in the scrollable lists. It can, however, modify the name of the file or even delete it. It works on the same basis as the instructions, borrowing from the command design pattern; all changes done by the user are performed on the songItems, and when the user chooses to export the changes, each songItem bears the responsibility of commiting these changes to disk.

A quirk about adding new items to already rendered sizers is that they are not immediately adjusted to the size of the parent, and are instead shown to the user with their natural size until a layout update is called, either naturally by the infraestructure or the program, or manually by us in order to prevent it. However, even if we do this call manually, the item is first shown in its natural size for a single cycle, and then immediately has its size adjusted. This

is uncomfortable and jarring to see. A quick work-around for this was to create the items with a width of 0, so even if there was this single-cycle delay in the logic of the program, it is invisible to the user.

No issue was found with the file explorer or the files provided during testing. We did however come upon a performance issue when trying to simultaneously load a thousand files into the program. The system froze for several minutes, and was only able to load a fraction of the files selected. This is due to the program running out of memory, since panels are a fairly expensive resource to keep in memory and render, and each loaded file is added to the interface with a panel with several widgets appended to it.

Since the program was still usable with fairly large batches of files, this problem was deemed not a priority and relegated as future work.

6.3 Prototype 3: Metadata I/O

A direct extension of the prior prototype, songItems are now initialised with values read directly from the file's metadata tags using the music-tag library. These values are presented within a text entry so the user can manually edit their values.

During testing, we found that exporting changes to metadata tags added virtually no execution time in comparison to only modifying the filename or deleting the file. Adding cover art did add a noticeable delay to the export process, but it was fairly manageable; using several hundred images barely added a few seconds of delay, which was still satisfactory.

6.4 Prototype 4: Automated instructions

To create the visual programming interface, each instruction was represented by a singular, self-contained graphical element, similarly to songItems. This new class that represents an instruction is called a logicItem.

As was previously explained in 4.2.1, each instruction has two parts: a generic toolbar, and a body that changes depending on the instruction. This class is built following the same principles we've used thus far, building a hierarchy of panels subdivided with sizers.

In order to make the body of the logicItem change, we first detach from the sizer and destroy the panel that is currently filling the second cell of the object's sizer, and then create and attach the new instruction.

Each logicItem contains the information necessary to work within the program: it performs the conditional check to see if its instruction should be performed or not for each file, it keeps a reference to the list of loaded files, it acts as an interface to execute the code contained within the body, and can detach and destroy itself from the scrollable list. Every time the item

is called upon to execute its instruction, all it does is check whether the conditional statement passes, and then pass the songItem in question to the callable method contained in the body.

The body of a logicItem has the sole purpose of acting as an input interface for the function it performs, and executing said function to any songItem it is passed. It has three parts:

- A readable name to differentiate each instruction.
- A panel-sizer hierarchy that can be inserted into the body of the logicItem.
- A function that affets the information on a songItem, typically using information gathered from the graphical interface part of the logicItem it belongs to.

In order to allow this program to be used as a platform by other users who want to create their own instruction, we abstracted the body of the logicItem into its own class, called "logic_module". If a user wants to create a new instruction, all they need to do is create a new class that derives from logic_module, and fill in the three elements that make it up. The program will then dynamically gather all available logic modules and add them as options in the combobox of the toolbar for the logicItems. Logic_module itself is derived from the wxpython panel class, so we call its constructor when changing the instruction of a logicItem and append it to the sizer. Then, when the time comes for the logicItem to execute its instruction, it passes one by one all loaded files in the form of songItems to the function hosted in the logic_module.

As a base to make the program functional, four logic modules were created along with the program with the following functions:

- Change the export option of the songItem to either commit changes, ignore the changes, or delete the item.
- Split a tag and distribute each half into another tag.
- Merge any combination of two tags plus a text entry into a single tag.
- Substitute any substring of text with the contents of a text entry (The lookup is done with regular expressions).

Five conditional statements can be used along with those four modules: a text based query to check if any of the supported metadata tags currently contains a given string of text. The provided statements support lookups with regular expressions, and can invert the logical output of the result. While there is not a simplified platform for developers to add new conditional checks, as is the case with logical modules, more could be added without compromising the integrity of the program.

At this point, all the technical requisites of the program are complete. Using the four provided logic modules we were able to fulfill all cases of use that were considered in testing while trying to emulate user behaviour. They were mostly comprised of regular pattern matching to split properly formatted filenames and fill missing metadata tags, but there were also extended tests to try to comply with more complex behaviours, such as fixes for capitalisation consistency, and removing naming artifacts derived from the mass-download of audio files, such as the date of production or quality denominators.

Chapter 7

Closing statement and future works

The application developed during this project fulfills all objectives set forth: to have an application that can repeat user-given instructions to a set of audio files, and to allow developers to add new instructions. In this regard, the project was successful.

If development of the application were to continue, the first step to take would be the abstraction of the songItem and logicItem classes so they can be used without being rendered on the screen. Not only would this solve the performance issue when dealing with very large batches of files, but it opens the possibility for the application to also be used as a command line asset.

Another handy feature could be the ability to export and import sets of instructions, though this addition could add a significant overhead to any user-created logic modules, since every logic module would need to define how their interior data should be represented as a standalone file and then reinterpreted back from said file.

Finally, since the music_tag library supports a wide arrange of audio file formats, it only makes sense to continue testing of the platform to ensure all suported files work properly and ensure the project is truly generic.

Bibliography

[1] *Tagmp3.online, a browser-based metadata tag editor. Available in https://tagmp3.online/.*

[2] *Mp3tag, a desktop application for batch metadata tag editing. Home page: https://www.mp3tag.de/.*

[3] *Wxpython, cross-platform GUI toolkit for the Python language. Home page: https://wxpython.org/.*

[4] *Music-tag, a library for editing audio metadata with an interface that does not depend on the underlying file format. Project available in https://github.com/KristoforMaynard/music-tag.*