CIS543 P1 Programmer's Documentation

Derek Strobel: dstrobel@cs.uoregon.edu

October 2019

Overview

This documentation gives an overview of the implementation details of the 5-button audiobook player system. It covers the following topics:

1. The Audiobook class
2. The AudibookPlayer class
3. Information Mode
4. Limitations

Acknowledgements

The following sources were used in the making of this project:

- regexr.com: an online utility used to interactively write regular expression strings
- notevibes.com: an online utility used to generate text-to-speech samples
- zamzar.com: an online file converter utility (used to convert mp3 to wav)

Packages

My program uses the Python Standard Library packages `os` and `re`. As these are standard Python libraries, they do not need to be installed. I have also written the following justifications for my use of these packages.

- `os`: This package is used simply to list the files in a directory. The P1 supplied code uses constants specifying the paths to files in the Norman book. I found this method unacceptable for loading files in a legitimate application. My method instead uses `os` to iterate over the files in a directory, parse their filenames, and appropriately organize them into the `Audiobook` data structure. Instead of using hard-coded literals, my solution allows an arbitrary book to be loaded as long as its files follow the file naming convention laid out in the problem specification.
- `re`: The regular expression package is simply used to parse filenames efficiently.

1 The Audiobook Class

The `Audiobook` class represents and organizes all of the information relevant to an entire audiobook. Given a path to a directory containing all of the sound files named according to the Project 1 filename specification, `Audiobook` loads the files and organizes them using a few connected data structures.

1.1 Chapters

The bulk of actual audio data in the `Audiobook` class is organized by chapter via the `AudiobookChapter` class. This is a small, simple data structure which simply holds all audio data pertinent to a given chapter. The `AudiobookChapter`s are all then indexed by chapter number into the `chapters` dict in `Audiobook`.

2 The AudiobookPlayer Class

`AudiobookPlayer` is the main high-level class of the application. It controls state management and holds all of the `Audiobooks` loaded into the program.

2.1 The Mode Stack

In this application, mode management is implemented as a stack. Each mode is a class which inherits from an abstract class, `AudiobookMode`, which implements some event handling functions: `on_enter` which is called when the state is entered, `on_exit` which is called when the state is exited, and `on_button` which is called when an action button is pressed.

`AudiobookPlayer` keeps track of the stack as a list, and controls mode pushes and pops. The stack system is designed to facilitate implementing back functionality.

Each mode is responsible for using methods from the `AudiobookPlayer` to manipulate the sound queues according to that mode's functionality. For example, the `PlayMode` enqueues the selected `Audiobook`'s sound files in order. The `PauseMode` clears the sound queue to silence system immediately.

As a result of the mode stack idea, this implementation leaves one key (;) always dedicated as a "back" key. This key simply pops the top mode (the current mode) off of the stack, leaving the previous mode as the new current mode and triggering any mode events which may occur on these transitions. This allows for a very modular state organization.

In addition, because each mode's button handling implementations dictate what happens on a button press from that mode, it is very easy to modify the state structure of the program in future updates to the program.

3 Information Mode

Information mode is a special mode which is accessible from any other mode using a consistent key (the Info key - L in the current implementation). It is pushed on top of the mode stack regardless of the source mode, so that the user can always use the back key to return to what they were doing prior to asking the system for help.

The purpose of the information mode is to supply a portable, consistent way for the user to access the information about the complete state of the program. The user can determine which book and which chapter they are reading from from any state.

4 Limitations

4.1 Granularity of Consumer Process Feedback

Unfortunately, the OSX implementation of the `JoinableQueue` from Python's `multiprocessing` library does not support a method for the producer process to check the progress of job completion by the consumer process (see `https://docs.python.org/2/library/multiprocessing.html#multiprocessing.Queue.qsize`).

As a result, I found it very difficult to glean any information about which audio files from the queue have been played, and thus any information while audio is playing which could be used to update the state of the system without modifying the `ks_play.py` implementation. Due to this limitation, it is not possible to skip backwards or forwards during playback. In addition, the highest granularity of state information that can be stored by the system about book progress is at the chapter level, because this is the lowest level which the user directly manipulates.