**MidiFind**

**Group 20**

Joe Crozier

Tyler Post

Victoria Bilbily

Version Number: 1.4

April 10, 2015

Software Engineering 2XB3: Binding Theory to Practice

Department of Computing and Software, McMaster University

# Revisions

April 10, 2015

Proofread by all members in the group. No further changes were made.

April 9, 2015

Added an internal review and evaluation of our design. Proofread and updated formatting.

April 7, 2015

Completed module descriptions and described the interface and semantics of each module implemented into the project. Finished uses relationship and documentation on implementation of the project.

April 2, 2015

Final design document constructed, title page, table of contents, and contributions recorded. Wrote executive summary and started on module descriptions.

*By virtue of submitting this document we electronically sign and date that the work being submitted by all the individuals in the group is their exclusive work as a group and we consent to make available the application developed through [CS] or [SE]-2XB3 project, the reports, presentations, and assignments (not including my name and student number) for future teaching purposes.*

Joe Crozier – 1311502 – Project Leader

*Project Specifications, GUI*

Tyler Post – 1302109 – Team Member

*Database, Design Documentation*

Victoria Bilbily – 1317465 – Log Admin

*Algorithms, Project Log Administrator*

# Contributions

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Role |  | Contribution |
| Joe Crozier  *Project Leader* | Project Specifications  Front End GUI |  | * Requirements Specifications Document * Project Presentation Slides * Created project graphics and artwork using Adobe Photoshop * Produced application using PyGame Library * Combined back-end parsing framework with search and sorting algorithms |
| Tyler Post  *Team Member* | File Parsing  Design Specification Document |  | * Developed an algorithm to decompose a midi file into a musical contour * Managed development of a database that can be efficiently searched * Produced original draft of Design Specification Document according to instructions received in SE 2AA4 |
| Victoria Bilbily  *Log Admin* | Searching and Sorting Algorithms  Project Log Administrator |  | * Implemented Knuth Morris Pratt string searching algorithm to effectively find matching contours * Optimized quicksort and three way quicksort in Python for mido song objects * Maintained living documentation of project and all deliverables * Organized and managed group meetings and schedules * Modified and finalized Design Specification Document |

# Executive Summary

MidiFind is a melody recognition tool designed to allow a user to easily input a melody and identify the track’s name and artist. Nearly anyone who enjoys music has experienced the frustration of knowing the melody of a song, but not the song’s actual name. MidiFind uses musical contours to identify different tracks, selecting from a dataset of 28,000 popular songs spanning all decades of music. A musical contour is a means of simplifying the melody of a song by expressing each note as a comparison to the previous note, stating whether the note is above (A), below (D), or the same (S) as the previous note. Musical contours are an excellent tool for identifying melodies, as they allow entire songs to be approximated with simple strings. By using musical contours rather than individual notes, the user does not have to input songs in their original keys, nor does their interpretation need to be exactly correct.

Table of Contents

[Revisions 2](#_Toc416460302)

[Contributions 3](#_Toc416460303)

[Executive Summary 4](#_Toc416460304)

[Design Overview 6](#_Toc416460305)

[Module Descriptions 8](#_Toc416460306)

[Class Descriptions 9](#_Toc416460307)

[FileParser 9](#_Toc416460308)

[MidiSong 12](#_Toc416460309)

[QuickSort 13](#_Toc416460310)

[StringSort 14](#_Toc416460311)

[GUI 15](#_Toc416460312)

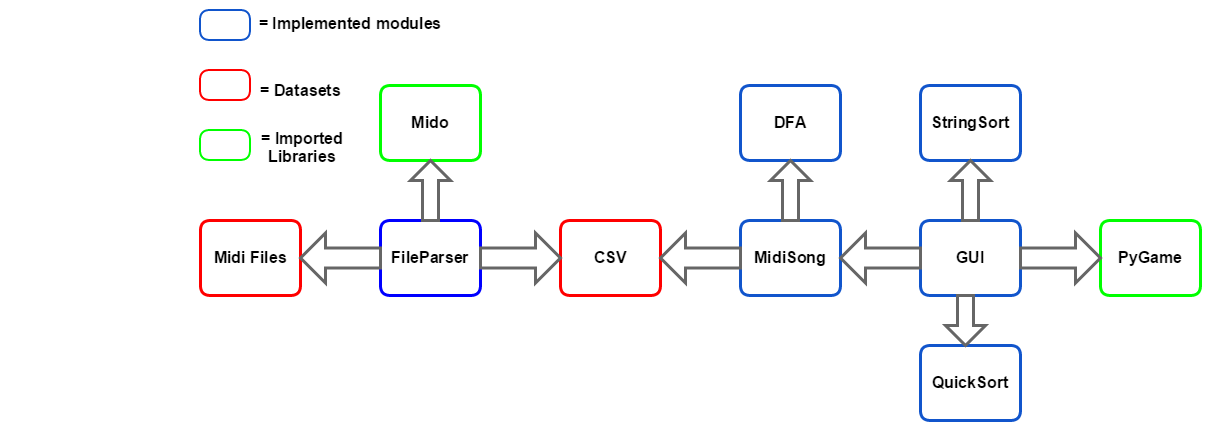
[DFA 19](#_Toc416460313)

[Requirements Trace-Back 21](#_Toc416460314)

[Testing 21](#_Toc416460315)

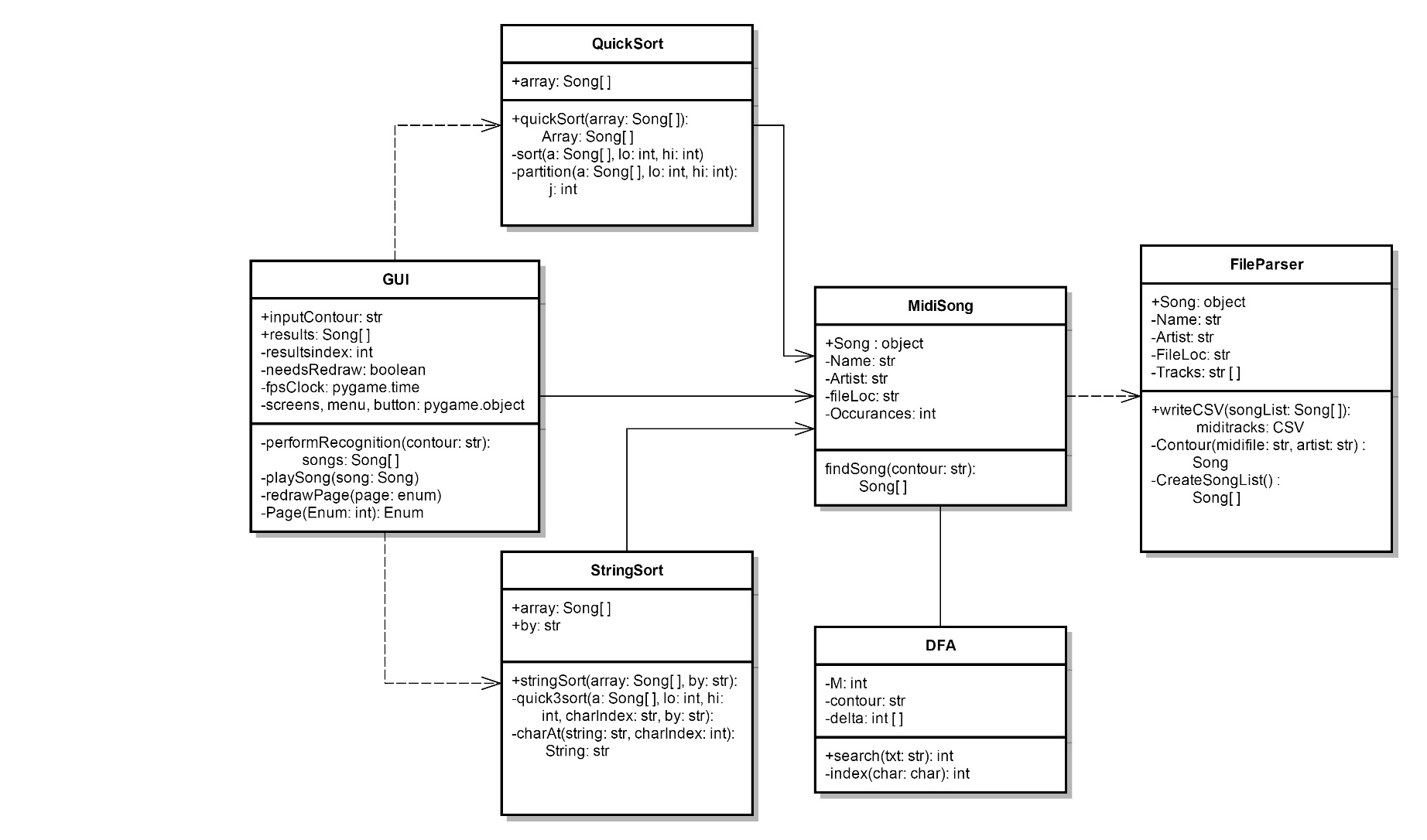
[Internal Design Review 23](#_Toc416460316)

# Design Overview



MidiFind is broken down into two major components, first creating a workable dataset, then searching that dataset for results. FileParser independently works with the midi files and uses the external library, Mido, to gather information from each song. FileParser then creates a csv with musical contours. After the csv is created, this module never has to be run again. Keeping the midi files separate from the searching application dramatically decreases run time.

Once the csv has been created, the application is run through the GUI class. This has been implemented with the external library PyGame. GUI calls MidiSong which handles reading the csv and calling DFA. This module separately searches for a set of characters in a large string, and returns the results. Once the matching songs have been passed back to GUI, StringSort sorts the results by artist or title, and QuickSort sorts the results by number of occurrences. By keeping our sorting algorithms separate they can be modified to stay generic, allowing the user to determine how they would like their results to be shown.

[MidiFind has been implemented in Python. This design decision was made because the success of our application is dependent on being able to retrieve all the musical information from the songs in order to construct a contour. Mido is a unique library that can extract note value, musical velocity, and time delay from a midi file. With the features from the Mido library and our efficiently implemented algorithms, we were capable of providing a working application that can match a small string input to a song from a database of thousands of songs. As PyGame is a dependency for Mido, it was decided this library should be used to create a workable GUI. ](NULL#_Toc416125218)

Unlike a program constructed in Java where each class is a file, Python is modularized. Each module is a unit of re-use, and accordingly we have implemented six modules to match this inherent design.  [These modules are internally independent and logically interact with each other, but it should be noted the specific class structure is intentionally distinct from a Java application.](NULL#_Toc416125218)

[Our submission includes source code, a database, and an executable. If the midi files are on the machine under MidiFind/Midi Files/\*, when a match has been found the song will play, however, due to the memory requirements of the song database, all but two midi tracks have been left out of submission. The application will still run using the contours located in the csv. It should be noted the algorithm requires a moderately unique contour of approximately ten notes, and will take up to twenty seconds to find possible matches. This is due to the vast number of notes inside our dataset.](NULL#_Toc416125218)

# Module Descriptions

**FileParser**

This module is responsible for creating musical contours for each midi track in our database and writing the results to a CSV. Doing a live search of each midi file is impractical and leads to very large run times. This module only needs to be run once, in order to gather the required information from the song to compose a contour.

FileParser uses the mido library to extract information about each note played in a song. Mido returns the value, velocity, and time of each note in the song, so it can be transcribed into a string contour. FileParser also handles the formatting of title and artist names, and finally creates a csv where each row contains a songs artist, name, and musical contour.

**MidiSong**

MidiSong is a module that is used to open and read the csv database constructed by FileParser. When a string is passed from the GUI it opens the csv and reads every line. FindSong constructs one DFA object for the given contour string and calls a search function from the DFA class for each line in the csv. If the string is found, it creates a Song object and saves the song name, artist, and file location. Once the entire database has been searched, it returns a list of Song objects to the GUI.

By implementing a separate module that opens and reads the csv, creating the contours is kept separate from searching the contours. At run time, this module eliminates the need to compile FileParser, saving memory and increases the speed of the searching algorithms.

**QuickSort**

This module is an optimized implementation of Quicksort, modified to sort Song objects by the number of matching contours in each track. If the user requests songs to be sorted by the number of matching contours in the song, GUI calls QuickSort with the list of matching songs which then returns a list ordered from highest to lowest. Before sorting the list, the module shuffles the elements in order to achieve a worst case Nlog(N) run time. Quicksort is a non-stable compare based algorithm which recursively divides the list into low elements and high elements until a base case is reached, at which point it returns the final sorted list.

**StringSort**

StringSort is a module that handles the ordering of song results that have returned a positive matching contour. The module is an implementation of three way quicksort for strings that returns an alphabetically sorted list in Nlog(N) time. GUI calls StringSort with a list of Song objects which is returned sorted by artist or title, which is determined by the user. This allows users to quickly find the desired song from a list of possible results.

**GUI**

GUI is the primary module in MidiFind that is responsible for handling input from the user and calling midiSong to search for positive matches. Upon receiving an input from the user (mouse click, key press, etc.), the GUI completely redraws itself. Once a list of matches has been returned, GUI calls QuickSort or StringSort to display the results to the user. This module handles displaying graphics to the user and playing midi tracks.

**DFA**

This class is an efficient implementation of the Knuth-Morris-Pratt string searching algorithm. A DFA object is created by midiSong using the given musical contour, and as the database is parsed, DFA’s search method is called to find matches. This is the primary algorithm implemented in MidiFind that matches song contours to the inputted contours. This optimized algorithm retains characters that have already been searched allowing to bypass previously matched characters. When a positive match has been found, DFA’s search method returns the number of occurrences of that contour in the song, which is saved and passed to the GUI.

# Class Descriptions

## **MODULE: FileParser**

Reads midi song files and creates a musical contour represented by a string, then stores the strings in a csv.

**INTERFACE**

**Uses:**

CSV

**Access Programs:**

writeCSV(songList: List)

Creates a CSV that contains the contour, artist, name, and file location of each Song object in songList.

**IMPLEMENTATION**

**Uses:**

Mido, csv, os

[*http://mido.readthedocs.org/en/latest/index.html*](http://mido.readthedocs.org/en/latest/index.html)

**Variables:** none

**Access Programs:**

writeCSV(songList: Song[])

Stores each song in songList into a csv file including name, artist, file location, and a contour for each track in the song. Formats the text before passing it into the CSV.

*with open(“midiTracks.csv”, “write”):*

*w = csv.writer()*

*for song in songList:*

*temp = [song.getArtist(), song.getName(), song.getfilLoc()]*

*temp.extend(track for track in song.getTrack())*

*writer.writerow(temp)*

**Internal Methods:**

Song(name, artist, fileLoc):

Implementation of an abstract data type for each song that is read. Contains attributes name, artist, file location, and tracks. Each attribute contains an accessor method that is called inside the class.

*def \_\_init\_\_(self, name, artist, fileLoc):*

*self.name = name*

*self.artist = artist*

*self.fileLocation = fileLoc*

*self.tracks = [ ]*

contour(midiFile: str, artist: str): Song instance

Creates a mido object with midiFile which is iterated over to create a melodic contour of the song. Returns a Song object with attributes name, artist, file location, and contour.

*song = Song(name, artist, midiFile)*

*midisong = mido.midifiles.MidiFile(midiFile)*

*for i, track in enumerate(midisong.tracks()):*

*Contour = “”*

*for s in track:*

*if s.type = “note\_on” and s.velocity != 0:*

*note = s.note*

*if (note > lastNote):*

*Contour += “u”*

*elif (note < lastNote):*

*Contour += “d”*

*elif(note == lastNote):*

*Contour += “r”*

*lastNote = note*

*song.addTrack(Contour)*

*return song*

createSongList(): List[]:

Opens the database of midi files and reads each file, calling contour on it. Every song object returned by contour is appending to a list. Once all the files in the database have been read, createSongList() returns a list of Song objects.

*for root, dirs, files in (“Midi Files”):*

*songList = []*

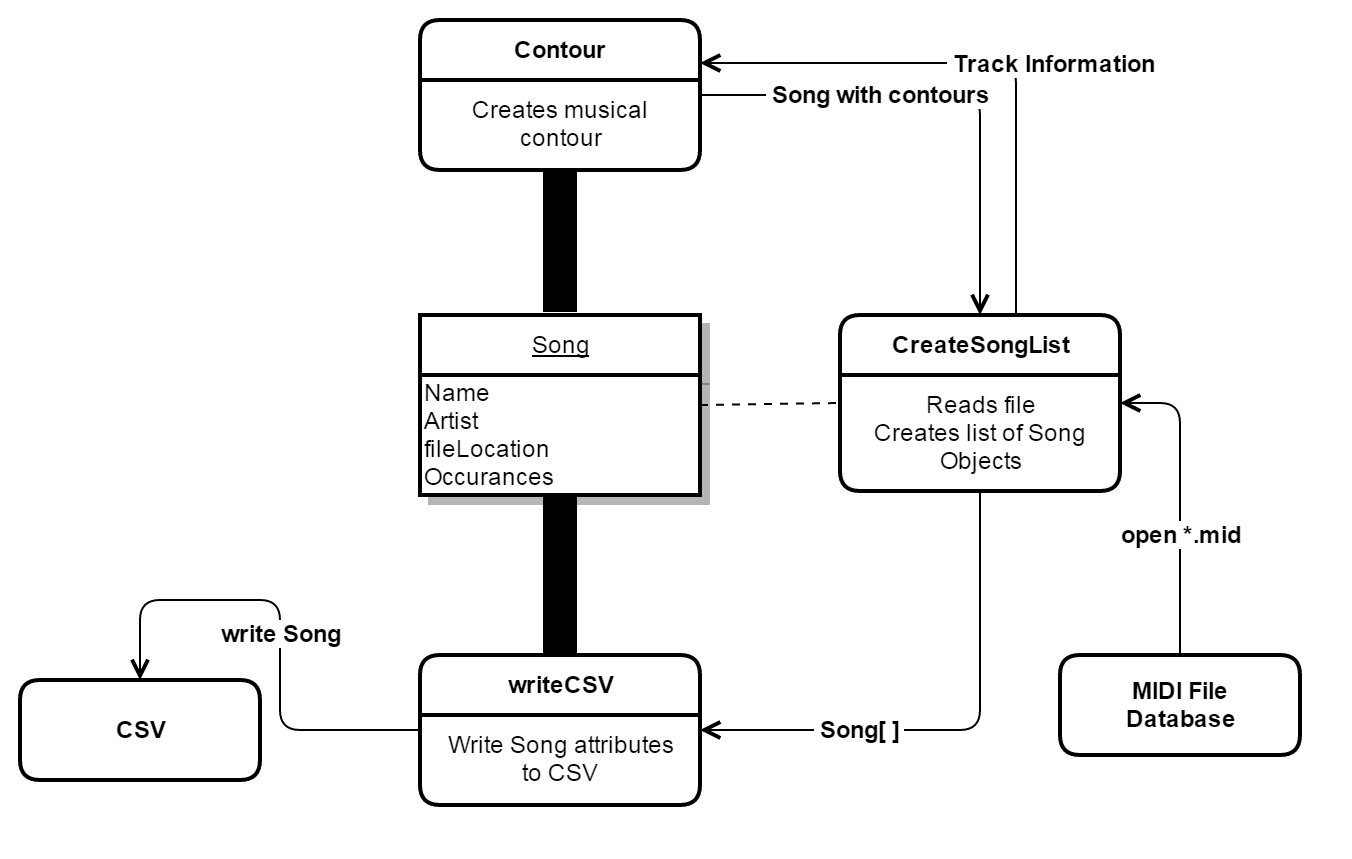
*for name in files:*

*artist = root.split(-1)*

*fileLoc = root + “//” + name*

*songList.append(contour(fileLoc, artist))*

*return songList*



## **MODULE: MidiSong**

Opens the database of contours and returns a list of matching Songs.

**INTERFACE**

**Uses:**

DFA

**Access Programs:**

findSong(contour: str): Song[]:

Opens a csv database and parses through every song track looking for the substring contour. Returns a list of Song objects if matches were made, otherwise returns None.

**IMPLEMENTATION**

**Uses:**

DFA, csv

**Variables:** none

**Access Programs:**

findSong(contour:str): Song[]

FindSong opens midiTracks.csv that was created by FileParser in order to search for the string contour. The searching for contour is handled independently by DFA which is called as FindSong parses through every contour. When DFA returns a positive match, a Song object is constructed and appended to the list of returned results.

*songsFound = []*

*with (open midiTracks.csv) as file:*

*contents = csv.reader(file)*

*dfa = DFA(contour)*

*for row in contents:*

*for i in range(3, len(row)):*

*if length(songsFound) == 50:*

*return songsFound*

*stringsFound = dfa.search(row[i], DFA)*

*if stringsFound > 0:*

*songsFound.append(Song(artist, title, fileLoc, stringsFound)*

*if length(songsFound) > 0:*

*return songsFound*

*return None*

**Internal Methods:**

Song(name, artist, fileLoc, occurrences):

Implementation of an abstract data type for each song that contains a matching contour. Contains attributes name, artist, file location, and occurrences of the inputted substring found in that track. Each attribute contains an accessor method that is found inside the class.

*def \_\_init\_\_(self, name, artist, fileLoc):*

*self.name = name*

*self.artist = artist*

*self.fileLocation = fileLoc*

*self.occ = occurrences*

## **MODULE: QuickSort**

Sorts an array of Song objects by number of melody occurrences.

**INTERFACE:**

**Uses:**

midiSong

**Access Programs:**

quicksort(a: Song[ ]): a: Song[ ]

Takes in an array *a* of Songs and sorts it by number of substring occurrences; also returns the array.

**IMPLEMENTATION:**

**Uses:**

midiSong, random

**Variables:** none

**Access Programs:**

quicksort(a:Song[]): Song[]:

Uses python’s random library to shuffle the Song array and minimize chances of O(n^2) running time. Calls upon the internal sort() method to sort the array by the instance variable *occ* and returns it.

**Internal Methods:**

sort(a: Song[], lo: int, hi: int):

Calls upon partition() to partition the array from *lo* to *hi* and recursively sorts both halves of the partition until the whole array is sorted (when *lo* and *hi* overlap.)

*if hi <= lo:  
 return  
else:  
 p = partition(a, lo, hi)  
 sort(a, lo, p-1)  
 sort(a, p+1, hi)*

partition(a: Song[], lo: int, hi: int): int

Partitions *a* from *lo* to *hi*. Takes the Song at index *lo* of the array to act as a pivot. Organizes the array so that every Song with a lesser *occ* value is to the pivot’s left, and every greater Song is to the pivot’s right. Returns the index of the pivot.

*pivot = a[lo].occ  
i = lo+1  
j = hi  
done = False  
while not done:  
 while i <= j and a[i].occ < pivot:  
 i ++  
 while i <= j and a[].occ > pivot:  
 j--  
 if j < i:  
 done = True  
 else:  
 a[i], a[j] = a[j], a[i]  
a[lo], a[j] = a[j], a[lo]  
return j*

## **MODULE: StringSort**

Sorts an array of Song objects alphabetically by either artist or title.

**INTERFACE:**

**Uses:**

midiSong

**Access Programs:**

stringSort(a, by):

Sorts array *a* by either artist or title, as indicated by the *by* argument.

**IMPLEMENTATION:**

**Uses:**

midiSong

**Access Programs:**

stringSort(a: Song[], by: str):

Calls upon quick3sort() to sort array *a* from 0 to the last index of *a* alphabetically by the attribute *by*.

I**nternal Methods:**

quick3sort(a: Song[], lo: int, hi: int, charIndex: int, by: str):

Sorts array *a* from *lo* to *hi* alphabetically by the char at *charIndex* of the Song attribute *by*. Uses 3-way partitioned quicksort for strings.

*if hi <= lo:  
 return  
lt = lo  
i = lo + 1  
gt = hi  
pivot = charAt(getattr(a[lo], by), charIndex)  
while i <= gt:  
 t = charAt(getattr(a[i], by), charIndex)  
 if t < pivot:  
 a[lt], a[i] = a[i++], a[lt++]  
 else if t > pivot:  
 a[gt], a[i] = a[i], a[gt--]  
 else:  
 i++  
quick3sort (a, lo, lt-1, charIndex, by)  
if pivot >= 0  
 quick3sort(a, lt, gt, charIndex + 1, by)  
quick3sort(a, gt+1, hi, charIndex, by)*

charAt(string: str, charIndex: int): int:

Uses python’s ord() method to return an integer ASCII value for the char at charIndex of string that may be used as an array index. Since alphabetical sorting is desired, with no care for case, all chars are converted to uppercase when determining their ASCII value.

*if charIndex < len(string):  
 return ord(string[charIndex].upper())  
else:  
 return -1*

## **MODULE: GUI**

Displays the user interface, connects other modules, and receives input from the user.

**INTERFACE**

**Uses:**

None.

**Access Programs**:

None.

**IMPLEMENTATION**

**Uses:**

PyGame, enum, DFA

<http://www.pygame.org/docs/>

**Variables:**

**fpsClock: pygame.time**

pygame Clock() object which regulates FPS.

**Screen: pygame.display**

pygame Screen object on which all content is blitted.

**menuPage**

Page enum which corresponds to the currently selected page.

**button1 – 3**: **pygame.sprite**

Button objects which update based on the selected page.

**Inputcontour: str**

String which contains the inputted contour

**Results: Song[]**

Results returned by MidiSong which contain inputcountour.

**Resultsindex: int**

Index of the currently presented Song object.

**needsRedraw: Boolean**

Boolean which when true instructs the GUI to redraw itself.

**IMPLEMENTATION**:

**Uses:**

MidiSong, pygame.locals, pygame.font, enum, sys.

**Access Programs**:

None.

**Internal Methods:**

performRecognition(self, contour): songs[]

Uses the passed string as a musical contour to call midiSong.findSong(), then sorts the returned Song[] array and returns the results.

*song = midiSong.findSong(contour)  
 if (NO\_SONGS\_FOUND or TOO\_MANY\_SONGS\_FOUND):  
 menuScreen = Failure  
 return  
 if (SORT\_BY\_OCCURENCE):  
 quicksort.quicksort(songs)  
 return songs  
 if (SORT\_BY\_ARTIST or SORT\_BY\_SONG):  
 stringSort.stringSort(songs, sorttype)  
 return songs*

playSong(self, song):

Uses the pygame mixer to initialize a music player and play the passed Song, based off of its fileLocation property.

*if (song.fileLocation):  
 pygame.music.play(song.fileLocation)  
 else:  
 print “Song not found.”*

*NOTE: This method only works if the passed song exists on the computer. Two MIDI files have been included with the app for testing sake, any other tracks can still be identified but will not play.*

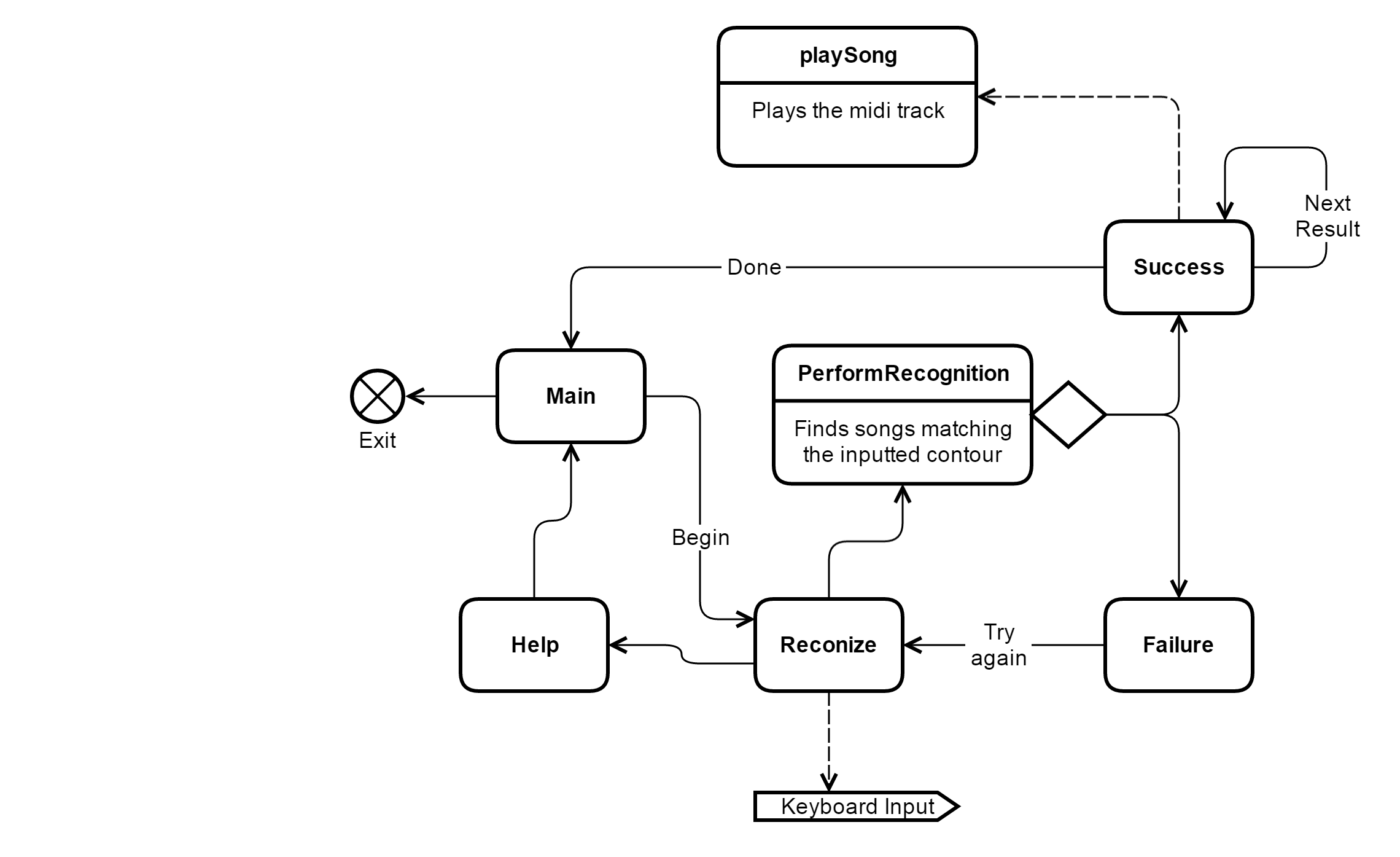
main(self):

Main loop for the program. 60 times a second, the method tests for input from the user in the form of pygame.Event objects. If a user clicks a button or presses a key, self.needsRedraw is set to True and the screen is blitted with updated content.

*while True:  
 if needsRedraw:  
 redrawPage(menuPage)  
 needsRedraw = False  
 for (event in Pygame.Events):  
 if (event.type = MOUSEBUTTONUP):  
 if (mouseLocation = button(1-4)location):  
 //PERFORM ACTION CORRESPONDING TO BUTTON  
 //i.e menuPage = Page.recognize  
 //i.e performRecognition(inputContour)  
 //i.e sorttype = “artist”*

*if (event.type = KEYUP and menuPage = Page.recognize):  
 if (event.key = A):  
 inputcontour = inputcontour + “A”  
 if (event.key = S):  
 inputcontour = inputcontour + “S”  
 if (event.key = D):  
 inputcontour = inputcontour + “D”*

*if (event.type = QUIT):  
 sys.exit()*



## **MODULE: DFA**

Creates a DFA object for a given pattern/contour. Implements Knuth-Morris-Pratt substring search methods for the DFA pattern within a given text.

**INTERFACE**

**Uses:** none

**Access Programs:**

\_\_init\_\_(pat):

Constructs a DFA representing the given contour/pattern.

search(txt): int:

Returns the number of occurrences of this DFA’s pattern in a given text.

**IMPLEMENTATION**

**Uses:** none

**Variables:**

M: int:  
 The length of the pattern, i.e. the index of the accept state.

Delta: int[][]:

A 2-dimensional int array representing the DFA transitions for a given contour or pattern of the {u/a,r,d} alphabet, using part of the Knuth-Morris-Pratt substring search implementation. The first dimension of the array represents the input character, and the second dimension represents the state transition.

contour: str:  
  
 The pattern/contour represented by the DFA object.

**Access Programs:**

\_\_init\_\_(pat: str): int[][]:

Instantiates variables M, contour, and delta. Delta is generated using part of the KMT algorithm.

*M = length(pat)  
delta = int[3][M]  
delta[index(pat[0])[0] = 1  
X = 0  
for j=1 to M-1, do:  
 for c=0 to 2, do:  
 delta[c][j] = delta[c][X]  
 delta[index(pat[j])[X] = j+1  
 X = delta[index(pat[j])][X]*

search(txt: str): int:

Counts the number of occurrences of the contour represented by the DFA using a modified KMP algorithm. The current state of *txt* is represented by *j*. If *j* reaches the final state of the DFA (in other words, if *j* equals the number of states in the DFA) then the occurrences counter is incremented. Returns the final value of the occurrence counter.

*N = len(txt)  
M = len(dfa[0])  
j = 0  
occ = 0  
for i=0 to N-1, do:  
 j = dfa[index(txt[i])][j]  
 if j == M:  
 occ ++  
 j = 0*

*return occ*

**Internal Methods:**

index(char: str): int:

Assigns integer values to characters so that they may be used as array indices.

*if char == ‘r’:  
 return 0  
else if char == ‘u’ or char == ‘a’:  
 return 1  
else if char == ‘d’:  
 return 2  
else:  
 print(char)  
 print(“Invalid character encountered!”)*

# Requirements Trace-Back

Each starred bullet is a requirement and each sub-bullet is a trace-back.

* The user may input a melody with no prior musical experience.
  + The user only needs to input a musical contour, which is the simplest representation of a melody possible. The GUI also has a “Help” screen.
  + The GUI module presents a text-box to the user, into which the user may insert A, S, and D characters to represent a musical contour. It also presents a “Recognize” button that may be clicked to begin the search
* Songs that contain the inputted melody will be searched for in a midi database.
  + The FileParser module parses a 28,000 midi song database into a CSV which contains each song’s musical contours, artist, and title.
  + To begin the search the GUI calls the midiSong module to iterate through each song in the CSV database and calls DFA on them to determine if the melody is present.
* Matched songs will be displayed to the user in order of relevance.
  + A list of matched songs is passed from MidiSong back to the GUI, which then calls upon methods from quickSort to sort the songs before the GUI displays them.
* These songs may also be sorted by artist or by title, for the user’s convenience.
  + The GUI presents buttons for different sorting displays, and calls upon methods from stringSort to perform the sorting before refreshing the output display.
* If there are no matches, the user will be informed and encouraged to try another input.
  + The GUI presents a “no matches found” page, including a restart button.
* If there are too many matches, the user will be informed and encouraged to try a more specific and lengthier input.
  + The GUI presents a “too many matches” age, including a restart button.
* Searches are fast and reproducible.
  + DFA has been optimized for MidiSong and tested for accuracy.
* The GUI is intuitive, attractive, and easy to use.
  + Joe Crozier is “GUI Joe” :^)
* The program is safe and reliable.
  + The program requires no personal data from the user. It uses little processing power and memory. It has been thoroughly tested for reliability.

# Testing

Testing was implemented thoroughly, using client code, unit tests, and descriptive testing instructions. Client code allowed for flexible and ongoing debugging during the development process. The unit test class algoTests, which used Python’s unittest library, was a more structured method of testing that, in the end, all of our algorithms work well enough to support the GUI. Finally, the GUI was used to test the back-end code with greater flexibility, and the GUI itself. These front-end tests were performed using the following instructions.

1. Action: Run GUI.py  
   Expected: A pygame window opens, displaying the main MidiFind screen. There are two buttons near the bottom of the screen: “Begin Recognition” and “Exit”
2. Action: Click “Begin Recognition”  
   Expected: The buttons on the screen are replaced with “Recognize,” “Help,” and “Cancel” buttons, and a text box appears for input. The caption displays instructions for the user.
3. Action: Click “Help”  
   Expected: The buttons on the screen will change to “Next” and “Back.” The caption will say “Need help?” and instructions will be displayed below it.
4. Action: Click “Next” twice.  
   Expected: New instructions appear each time it is clicked. On the second click, the button changes to “Done”
5. Action: Click “Done”  
   Expected: Returns to the screen described in step 2).
6. Action: Try to input any characters other than A, S, or D.  
   Expected: No characters appear in the text box.
7. Action: Input “sssssdd” and then click backspace twice.
8. Expected: The textbox will display “SSSSS”
9. Action: Click “Recognize”  
   Expected: The user is instructed to input a more specific contour, and previous buttons are replaced with a “Start Over button”
10. Action: Click “Start Over”  
    Expected: Returns to the screen input-contour screen described in step 2).
11. Action: Click “Back”  
    Expected: Returns to the main screen.
12. Action: Click “Begin Recognition”  
    Expected: Displays the input-contour screen.
13. Action: Input “dsaadsaaasddaddd” and click “Recognize”  
    Expected: Within 15 seconds, the screen displays something along the lines of “Hey Jude by The Beatles” and a midi version of the song plays.
14. Action: If there is a “Next” button, click it until “Done” appears, then click “Done.” Otherwise, just click “Done” immediately.  
    Expected: If there are more matches, they are displayed. When “Done” is clicked, the main screen is displayed again.
15. Action: Repeat steps 12-14, but in step 13 click “Sort Results By:” once before clicking “Recognize.”  
    Expected: The matches are displayed in alphabetical order by artist.
16. Action: Repeat steps 12-15, but in step 13 click “Sort results by:” twice before clicking “Recognize.”  
    Expected: The matches are displayed in alphabetical order by song name.
17. Action: Repeat steps 12-16 with at least 5 songs from midiTracks.csv. Input any reasonably unique and at least 12 character long portion of the song’s musical contour, replacing ‘r’ with ‘s’ and ‘u’ with ‘a’, and click “Recognize.”  
    Expected: For each search, one of the given outputs matches the input song. The outputs are sorted as they should be.
18. Action: Click “Back,” then click “Exit.”  
    Expected: The pygame window closes.

# Internal Design Review

We have created an application in Python that matches an inputted musical contour to songs that have this melody. In order to help users who can recognize but not identify a song, MidiFind offers a solution that matches a simple inputted contour to possible songs.  MidiFind is capable of handling large datasets and using various searching and sorting algorithms to find matching contours.  Quicksort, KMP substring search, and three way string quicksort are all used in our application and provide efficient results.  Once the user’s input has been matched to a song, the artist and name is displayed while the song is played through the user’s speakers.  If more than one song has been matched, the user may scroll through the list of songs containing the inputted contour.  All of these implementations match our design requirements.