LyricsByU

Version#: 1

Group#: 8

Team Name: Lyricist

Members:

Evan Ansell

Olivia Tinios

Tongfei Wang

Bowen Yuan

Tim Zhang

April 12, 2017

SFWR ENG 2XB3

McMaster University

Contents

1 Revisions

1.1 Revision History

Revision#	Date	Comments
1	14/03/2017	First prototype. Program gets one keyword as input from user through console and prints all lines from the dataset containing the keyword to the screen.
2	16/03/2017	Update: program now has GUI and no longer gets input through the console.
3	22/03/2017	Update: program now asks user to choose a genre and only searches for lines in songs that have the matching genre.
4	02/04/2017	Update: program now uses a graphing algorithm to choose lyric lines.
5	12/04/2017	Final prototype. Bugs fixed and comments added.

1.2 Roles and Responsibilities

Last Name, First Name	Student Number	Role in the Project
Ansell, Evan	1415992	Tester, File Processor(reads input from dataset)
Tinios, Olivia	400034007	Sort Algorithm Programmer (handles implementation of sorting algorithm)
Wang, Tongfei	001437618	Data Architect (handles ADTs)
Yuan, Bowen	400005913	Graph Algorithm Programmer (handles implementation of graphing algorithm)
Zhang, Tim	400006216	Search Algorithm Programmer (handles implementation of searching algorithm), Log Admin, Project Leader

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 applicationdeveloped through [CS] or [SE]-2XB3 project, the reports, presentations, and assignments (not including my name and student number) for future teaching purposes.

2 Contributions

Name	Contributions	Comments
Ansell, Evan	- Initial project plan (Milestone 1) - Stakeholders, system test procedures (Requirements Specifications) - Initial and final implementation of ReadFile class (Prototype) - ReadFile and ReadWord API (Design Specifications) - Managed our Git repository - Unit testing	- Edited and polished documents, especially the Requirements Specifications document. Managed Git repository to help with organization.
Tinios, Olivia	- Final project plan (with sprints) (Milestone 1) - Methodology, roles and modificaiton (Milestone 1) - Functional requirments (Requirements Specification) - GenreBinarySearch and MergeSort (Prototype) - GenreBinarySearch, BinarySearch and MergeSort API (Design Specifications) - Cover and revision pages, UML class diagram (Design Specifications) - Recorded meeting minutes and kept track of Scrum product backlog	- Edited documents and made sure that team was following Scrum methodology (organized sprints).
Wang, Tongfei	- Scope (Milestone 1) - Non-functional requirments (Requirements Specifications) - WordADT, Dictionary, SongADT and bug fix (Prototype) - WordADT MIS, SongADT MIS, SearchKeyword API and Dictionary API (Design Specifications)	- Provided critical thinking and implemented it in code.
Yuan, Bowen	- Objective (Milestone 1) - Corrective,adaptive,pecfective maintenance (Requirements Specificaitons) - ReadFile class, graph algorithm, interface design, bug fix (Prototype) - UInterface and Graph API (Design Specifications)	- Made a bunch of technical breakthroughs, especially with graphing algorithm.
Zhang, Tim	- Abstract (Milestone 1) - Domain (Requirements Specifications) - BinarySearch(Prototype) - UML state machine diagrams (Design Specifications) - Contribution page and executive summary (Design Specifications) - Presentation Slides - Updated the project log	- Assigned tasks and kept the project on track.

3 Executive Summary

Music occupies a great part of many people's leisure time, and lately it's become much more so with a rising use of portable devices and headphones. The music that is being produced by the industry's many talented artists brings inspiration and joy to the daily life of its listeners. *LyricsByU* provides a new way for music-lovers to create their own songs. Using a keyword entered by the user, *LyricsByU* uses lyrics from a dataset of 380,000 songs to make a new combination of lyrics. Users can also tailor the song to align more with their music preferences by specifying a genre, year or artist. With *LyricsByU*, anyone can create a song.

For this version of *LyricsByU*, only 3000 songs were used from the full dataset. These 3000 songs contained around 100,000 lines which fulfilled our functionality requirement. This version of *LyricsByU* also only allowed the user to specify a genre. The program could be improved in the future by allowing the user to specify the year and artist as well.

4 Description of Classes/Modules

4.1 Overview of Classes

Our program is made up of 11 classes: Song, Word, Dictionary, BinarySearch, GenreBinarySearch, MergeSort, ReadFile, ReadWord, Graph, SearchKeyword and UIinterface. Below is a brief description of each class.

- Song : An ADT that represents a song(contains artist, lyrics, genre, year released, etc...)
- Word : An ADT that represents a word from a song (contains the location of the word i.e. what song/line it came from).
- Dictionary : An ADT that represents Dictionary containing all words from the dataset.
- ReadWord : Puts all the words into three different Dictionaries.
- Graph : Connects two lyric lines with same word.
- BinarySearch : Searches for the location of a word in a dictionary.
- Genre Search : Finds all songs that match a given genre in a list of songs.
- SearchKeyword: Searches for a specific keyword using binary search.
- Mergesort : Sorts a list of songs by genre (in alphabetical order).
- Readfile : Reads input from the dataset.
- UIinterface: Creates the user interface for the program.

Creating an ADT for the songs in our interface allows us to keep track of all the information associated with a song (genre, artist, lyrics, etc...). Our Dictionary class allows us to easily search through a large number of words for a specific keyword and our word ADT allows us to store the location of each word (i.e. what song/line it came). Our sorting, searching and graphing algorithms are separated into MergeSort, BinarySearch, GenreBinarySearch and Graph. Two classes are required for binary search because each one serves a different purpose. We also have a ReadFile class which handles input from the dataset and a UIinterface class which handles output and displays the created song.

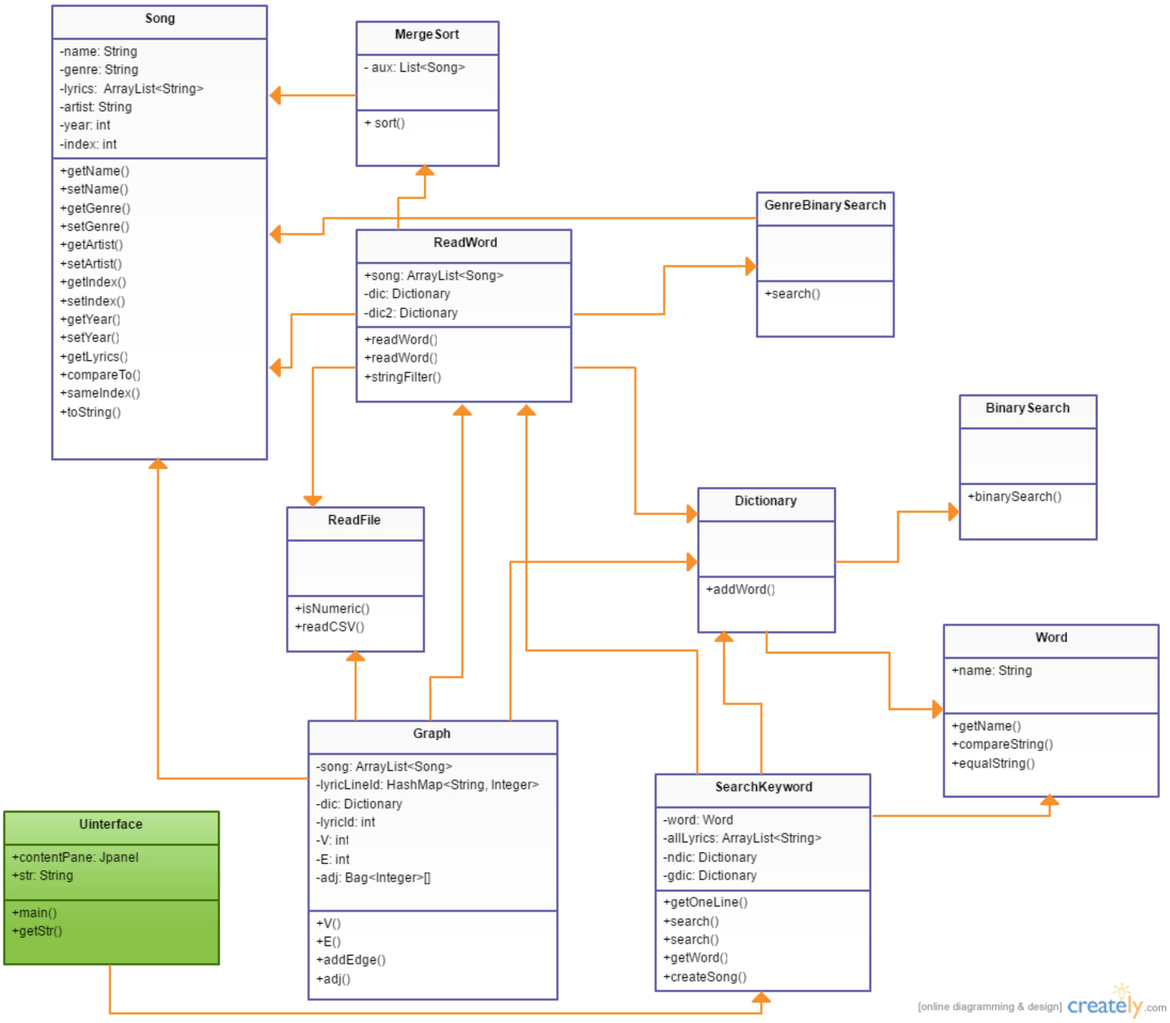

Figure 1: UML Class Diagram

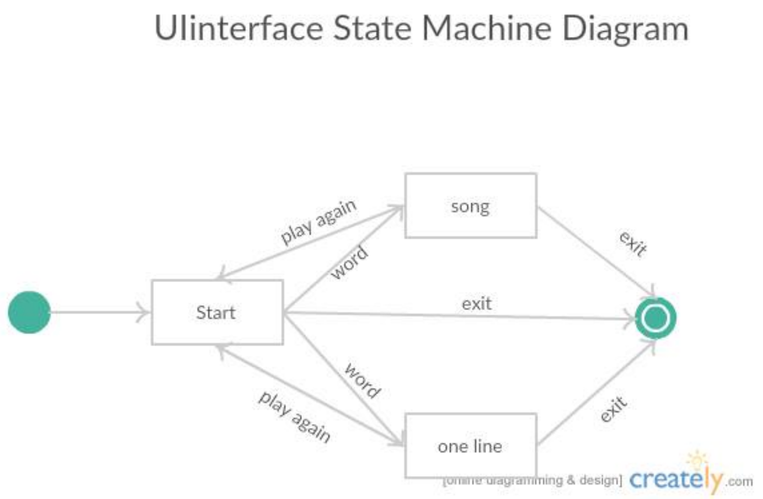
UIinterface State Machine Diagram
Start
song
one line
play again
word
exit
word
play again
exit
exit
[online diagramming & design] creately.com

Figure 2: UML State machine for UIinterface class.

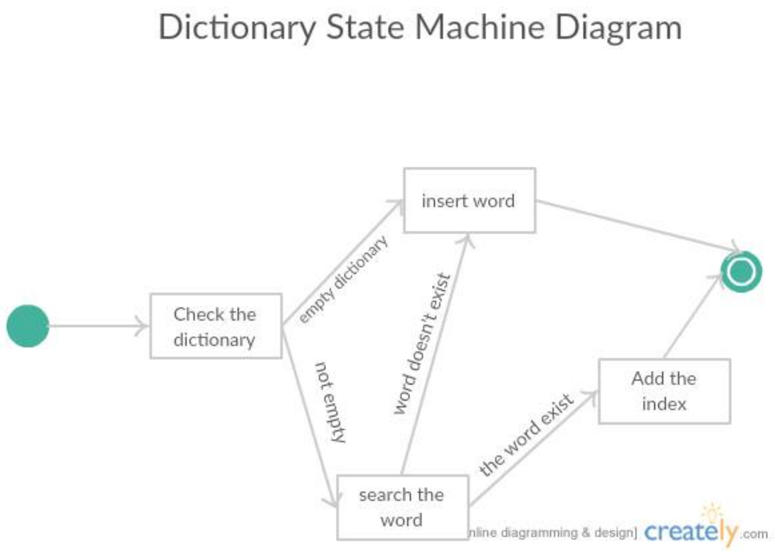

Figure 3: UML State machine for UIinterface class.

4.2 Song ADT

Template Module

Song

Uses

N/A

Syntax

Exported Types

Song = ?

Exported Access Programs

Routine name	**In**	**Out**	**Exceptions**
new Song		Song	
getName		String	
getGenre		String	
getArtist		String	
getLyrics		String	
getIndex		int	
getYear		int	
setGenre	String		
setArtist	String		
setYear	int		
setIndex	int		
setName	String		
compareTo	Song	int	
sameIndex	Song	boolean	
toString		String	

Semantics

State Variables

name: String
genre: String
lyrics: sequence of String
year: int
index: int

State Invariant

None

Assumptions

None

Access Routine Semantics

new Song ():

- transition: $lyrics = <>$
- output: $out := \mathit{self}$
- exception: none

getName():

- output: $out := self.name$
- exception: none

getGenre():

- output: $out := self.genre$
- exception: none

getArtist():

- output: $out := self.artist$
- exception: none

getLyrics():

- output: $out := self.lyrics$
- exception: none

getIndex():

- output: $out := self.index$
- exception: none

getYear():

- output: $out := self.year$
- exception: none

setYear(y):

- transition: $self.year := y$
- exception: none

setGenre(genre):

- transition: $self.genre := genre$
- exception: none

setArtist(artist):

- transition: $self.artist := artist$
- exception: none

setIndex(i):

- transition: $self.index := i$
- exception: none

setName(name):

- transition: $self.name := name$
- exception: none

compareTo(other):

- output: $out := (self.index > other.index \Rightarrow 1)|(self.index < other.index \Rightarrow$ -1)
- exception: none

sameIndex(other):

- output: $out := (self.index = other.index) \Rightarrow true$
- exception: none

toString():

- output: $out :=" --- Song : " + self.name + "Artist : " + self.artist + "Year : " + self.year + "Genre : " + self.genre$
- exception: none

4.3 Word ADT

Template Module

Word

Uses

N/A

Syntax

Exported Types

Word = ?

Exported Access Programs

Routine name	In	Out	Exceptions
new Word	String, int, int	Word	
getName		String	
compareString	String	int	
equalString	String	boolean	

Semantics

State Variables

name: String

State Invariant

None

Assumptions

None

Access Routine Semantics

new Word $(name, songIndex, sentenceIndex)$:

- transition: $self.name := name$
- output: $out := self$
- exception: none

getName():

- output: $out := self.name$
- exception: none

compareString(other):

- output: $out := (self.name > other.name \Rightarrow 1)|(self.name < other.name \Rightarrow -1)|(self.name = other.name \Rightarrow 0)$
- exception: none

equalString(other):

- output: $out := (self.name = other.name) \Rightarrow true$
- exception: none

4.4 MergeSort

The MergeSort class is used to sort a list of songs by genre (in alphabetical order).

public class	MergeSort	
void	sort(ArrayList$< Song >$ x)	*Creates auxiliary list (elements will be copied into this list during merge) and calls the other sort method.*
void	sort(ArrayList$< Song >$ x, int lo, int hi)	*Recursively sorts sub lists and then combines them using the merge method.*
void	merge(ArrayList$< Song >$ x, int lo, int mid, int hi)	*Merges two sub lists.*
boolean	less(Song v, Song w)	*Returns true if* $v < w$.

4.5 GenreBinarySearch

The GenreBinarySearch class is used find all songs in a list that match a given genre.

public class	GenreBinarySearch	
int[]	search(ArrayList$< Song >$ x, String s)	*Returns array with low and high indices.*
int	lowIndex(ArrayList$< Song >$ x, String s, int lo, int hi)	*Finds matching element with lowest index.*
int	highIndex(ArrayList$< Song >$ x, String s, int lo, int hi)	*Finds matching element with highest index.*

4.6 UIinterface

The UIinterface class creates the GUI.

public class	UIinterface	
	UIinterface	*Creates user interface.*
String	getStr()	*Gets user input.*

4.7 ReadFile

The ReadFile class is used to read the dataset.

public class	ReadFile	
boolean	isNumeric(String str)	*Reads a comma-separated file of songs. Song objects containing information on the songs (index, song name, year, artist, genre, lyrics) are created for each unique song in the file and added to an ArrayList of type Song which is returned.*
ArrayList< *Song* >	readCSV()	*Is given the first "word" from each line and checks if it is a number. If it is a number then this indicates that the line being read is likely the information for a new song.*

4.8 ReadWord

The ReadWord class is used to generate a dictionary of words.

public class	ReadWord	
Dictionary	readWord()	*Generates Dictionary (essentially a special ArrayList of Word objects) from all songs.*
Dictionary	readWord(String genre)	*Generates Dictionary from all songs of the specified genre.*
String	stringFilter(String str)	*Removes special characters (e.g. !;:&̂) from the given string.*

4.9 BinarySearch

The BinarySearch class is used to find a word in a dictionary (an ArrayList of words) that matches the given keyword.

public class	BinarySearch	
int	binarySearch(Dictionary words, String value, int min, int max)	*Returns index of matching word in Dictionary.*

4.10 Graph

The Graph class is used to find lines from the lyrics dataset with matching words. Each vertex is a lyrics line and lines containing the same word are connected.

public class	Graph	
	Graph()	*Give each lyric line index an independent lyricId, and get every word in dictionary, connecting each pair of lyric line indices(lyricId) that have that word.*
int	V()	*Returns number of vertices.*
int	E()	*Returns number of edges.*
void	addEdge()	*Adds an edge when two lyric lines contain the same word.*
Iterable< *Integer* >	adj(int v)	*Returns all vertices adjacent to vertice v.*

4.11 SearchKeyword

The SearchKeyword class searches for lines in the dataset (using the keyword) and creates a song.

public class	SearchKeyword	
String	getOneLine(Dictionary dic, String keyword)	*Returns a line containing the keyword from the lyrics dataset.*
String	search(String keyword)	*Searches for a line in the full dictionary.*
String	search(String keyword, String genre)	*Searches for a line in the genre restricted dictionary.*
Word	getWord()	*Returns a Word.*
String	createSong(String keyword)	*Finds lines and creates a song.*

4.12 Dictionary

The Dictionary class is an ArrayList of Word objects. It stores all words from the lyrics dataset.

public class	Dictionary	
void	addWord(String word, int songIndex, int sentenceIndex)	*Adds a word to the dictionary.*

5 Design Evaluation

At the end of our project's development cycle, our group has developed a program which meets most but not all of our goals and requirements.

In its current state of implementation it does not fulfill our requirement #5 but instead simply stops generating a song at a predetermined length. It also only fulfills part of requirement #4, but the ability to choose lyrics from year or artist could be easily added on in future releases. Additionally, our program is currently only using a small sample of the dataset (∼100,000 lines vs 10,000,000) due to the immense amount of processing time and memory required to handle the full dataset. If the algorithms and interactions between the classes of our program were optimized, it might allow us to use a more complete version of the dataset.

Our program was developed using the Scrum methodology. The methodology worked well for us and setting short term goals allowed us to focus on the tasks at hand without getting side-tracked by other aspects of the project. Also, planning the dates for each sprint in advance helped keep us on track and helped prevent us from falling behind. Our development process could have been improved by having daily meetings instead of meeting only at the beginning and end of each sprint. Sometimes it was unclear which tasks were assigned to which person and checking up daily on everyone's progress could have helped to fix this and improve overall productivity.