MesosticMachine:
A Software System to Implement Algorithms Specified in a Musical Score by John Cage

A dissertation submitted in partial fulfilment of
the requirements for the degree of
MASTER OF SCIENCE in Software Development
in
The Queen's University of Belfast
by

Martin W. Dowling

13 September 2017

Declaration of Academic Integrity

Before signing the declaration below please check that the submission:

1. Has a full bibliography attached laid out according to the guidelines specified in the Student Project Handbook?
2. Contains full acknowledgement of all secondary sources used (paper-based and electronic)
3. Does not exceed the specified page limit
4. Is clearly presented and proof-read
5. Is submitted on, or before, the specified or agreed due date. Late submissions will only be accepted in exceptional circumstances or where a deferment has been granted in advance.
6. Software and files are submitted via Subversion.
7. Journal has been submitted

I declare that I have read both the University and the School of Electronics, Electrical Engineering and Computer Science guidelines on plagiarism - http://www.qub.ac.uk/schools/eeecs/Education/StudentStudyInformation/Plagiarism - and that the attached submission is my own original work. No part of it has been submitted for any other assignment and I have acknowledged in my notes and bibliography all written and electronic sources used.

I am aware of the disciplinary consequences of failing to abide and follow the School and Queen's University Regulations on Plagiarism.

Name: (BLOCK CAPITALS) MARTIN W DOWLING

Student Number: 10013075

Student's signature

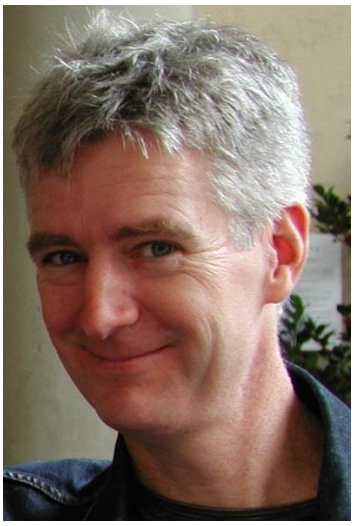
Date of submission: 13 September 2017

Acknowledgements

It is a pleasure and an honour to record my sincere gratitude to Dr Ian O'Neill for his attentive supervision of this thesis, Aidan McGowan for tuition and advice, Úna Monaghan inspiring me , collaborating with me, and advising me, and Christine Dowling for her inexhaustible support and understanding.

Abstract

This dissertation develops and implements a software system which automates significant elements of a score written by the artist and composer John Cage. The score comprises a set of instructions to transform the text of a book into a musical performance. The dissertation transforms a number of these instructions into the requirements specification for a software system, describes the key functions of that system, illustrates the design of the system, and provides details of the implementation of the system using an object oriented design methodology with the Java programming language in the Eclipse IDE. The Python programming language is also used to query the Natural Language ToolKit to query the WordNet database at Princeton University.



Martin Dowling

Table of Contents

Introduction

In the 1970s the composer and artist John Cage developed techniques for translating texts of literary works into chance determined poetry and using these to translate the works into performances. He visited Ireland and in collaboration with Irish musicians created and produced a number of performances of a piece titled *Roaratorio, An Irish Circus On Finnegan's Wake*. In 1992, the year Cage died, a double compact disc was released entitled *Cage: Roaratorio; Laughtears—Cage, Shöning, Heaney, Ennis, et. al; Writing for the Second Time Through Finnegan's Wake*.[1] This recording is well known to Irish avant garde artists, performers, and traditional musicians because of its use of traditional musicians to translate James Joyce's text into a musical performance. At some point Cage wrote a retrospective score which included the full text of the score for this performance, and this score was included in the booklet accompanying 1992 compact disk. Cage gave the score the following title:

"_____ , _____ _____ Circus On _____"
(*title of composition*) (*article*) (*adjective*) (*title of book*)

The score comprised seven paragraphs of instructions which were, as its subtitle reads, "a means for translating a book into a performance without actors, a performance which is both literary and musical or one or the other." In 2012 the present author, in collaboration with musician and sound engineer Úna Monaghan and the poet and writer Ciaran Carson, applied this score to Carson's memoir *The Star Factory*,[2] Carson's memoir of his Belfast childhood, and produced three performances of the piece under the title *Owenvarragh: A Belfast Circus On The Star Factory*. Two of these performances took place in the Sonic Lab at Queen's University of Belfast, and a third at a Festival of Ideas in at the University of York.[3] Dowling and Monaghan then published a detailed account of the development of this performance.[4] In that publication Cage's score is summarised as follows:

> (1) choose a book and obtain permission for its use; (2) create a chance determined text based on the book and record and time a recitation of this text; make lists (3) of

[1] Mode Records, 28/29, New York, 1992.

[2] Ciaran Carson, *The Star Factory* (Granta, London, 1997).

[3] A video documentary recording of the first performance is available on martin YouTube: www.youtube.com/playlist?list=PLOa0O4aBIs3UPWfV1l-A0WKO25866Duzz.

[4] Martin Dowling and Úna Monaghan, "Creating a Circus of Words, Music, and Sound: From Roaratorio to Owenvarragh" *Journal of Black Mountain College Studies*, Special John Cage Issue, vol. 4, 2013 (www.blackmountainstudiesjournal.org). For a more complete account of this work in the context of Cage's art and influence, see Úna Monaghan, *New technologies and experimental practices in contemporary Irish traditional music: a performer-composer's perspective* (PhD, Queen's University of Belfast, 2015), chapter 2.`

> the places and (4) of the sounds mentioned in the book; (5) collect recordings of these places and sounds and, using time points in the recitation and page/line locations in the original text as guides, mix the recorded sounds with the recitation into a multi-track recording; (6) record or perform "relevant musics" by soloists and superimpose these onto the multi-track recording; (7) realise the piece as a stereo recording, a multi-channel recording, or a performance with the layers created in (2) and (6) performed live.[5]

To follow these instructions and create a performance involves a number of extremely labour intensive, time-consuming, and error prone task tasks. Very few artists, aside from Cage himself, have fully completed a project based on the score. The prohibitive scale of the tasks Cage laid out in this score may well explain why very few other works have come to light aside from *Roaratorio* and *Owenvarragh*. Working with eye and hand to complete the tasks for extracting and organising textual material from *The Star Factory* in steps (2), (3), and (4) above took the better part of a month of full-time, meticulous labour. The idea of developing an automated process for selecting words and organising them into the poetic form presents itself immediately to anyone involved in this kind of meticulous labour.

One goal of this project is to make some of these more arduous tasks easier to accomplish. The goal is to create a system which, given the appropriate electronic formatting of the target text, will perform the necessary tasks for steps (2), (3), and (4) in a fraction of the time required to do so by eye and hand, providing artists with preparatory textual materials so that they can move directly onto the remaining tasks of recording a recitation, collecting sounds and creating a multitrack recording for a performance. If this project succeeds in raising awareness of Cage's score and encouraging artists to make use of this system to bring new performances of literary works to fruition, then the effort will have been justified.

A second goal of this project is to provide functionality that produces more accurate material than working by eye and hand through a text. Dowling and Monaghan offered the following reflection on their experience of the untrustworthiness of the eye and hand.

> It was one thing to correct syllables not properly excluded, but another to ensure that words were not missed that should have been included earlier than they were eventually to be. Martin accomplished the former, after many hours, but abandoned the latter as futile, or perhaps as an exercise in "purposeless purposelessness" beyond the limits of his endurance. In the end, we included human error in the grand strategy of chance determination. At an early stage we considered and rejected the idea of developing an automated process for selecting words, writing a program

[5] Dowling and Monaghan, "Creating a Circus," p. 4. The entire score is reproduced in Appendix 1 of this document.

> which follows Cage's instructions using optical character recognition of the text. We much preferred the tactile and personal experience offered by Cage's way of 'reading through.'[6]

If this project succeeds in providing a system that more accurately follows Cage's instructions for dealing with a text than the eye and hand method used in *Owenvarragh*, the effort will have been doubly justified.

This dissertation gives an account of the development of such a system, called *MesosticMachine*. Its structure adheres to the template specified in the handbook for the CSC7057 Individual Software Development module on the MSc in Software Development at Queen' University of Belfast, and contains the following chapters:

- Chapter 1 describes the problem of developing a software system to assist users in the creation of a performance of Cage's score. It explains this problem domain by defining a set of problems to be solved, noting critical issues that arise for the translation of the current "hand and eye" approach to a project of this nature into software functionality. This chapter is based in part on the original problem specification document submitted in June 2017 in accordance with the requirements of the CSC7057 Handbook.
- Chapter 2 follows directly on from Chapter 1 by proposing solutions to the problem set. A strategy for developing the system is then articulated.
- Chapter 3 offers a requirements analysis and specification for the software system. It explains how the requirements were elicited and interpreted, identifies various users and use cases for the software system, defines the core functions of the system to be developed.
- Chapter 4 illustrates the design of the system using sequence and class diagrams, linking the design to the specification defined in Chapter 3.
- Chapter 5 drills down to the actual development environment, programming language, and coding techniques used to bring the system to actual implementation and functionality. The chapter explains how principles of object oriented programming guided the implementation strategy, and walks through each of the dozen interacting objects that comprise the system, providing explanation and code snippets for particularly interesting solutions.
- Chapter 6 offers an evaluation of the system in its current state of development at the time this dissertation was submitted, identifies the most and least successful aspects of the project, and makes suggestions for further development.

[6] Dowling and Monaghan, "Creating a Circus", p. 9.

Chapter 1: Problem Specification and Domain Analysis

This chapter expands on an earlier problem specification document submitted in June 2017 in accordance with the requirements of the CSC7057 handbook. The problem this project seeks to solve, in a nutshell, is to convert, where possible, the algorithms specified in the score for dealing with the text of a literary work into computer algorithms, and organise these into a fully functional software system with a graphical user interface. This interface has its specific target audience artists and sound engineers who might be interested in creating a performance using Cage's score but are inhibited by the scale of the task. It also targets musicians and other performers involved in a such a performance, and potential publishers of the texts generated by the implementation. The goals of the project are to at a minimum provide good working functionality to create accurate mesostics, lists of places, and lists of sounds. A simple and straightforward graphical user interface, serving to receive the necessary parameters (in the form of formatted text files), start the required processes, and produce useful text file outputs, is envisioned.

The problem domain comprises the first four of the seven paragraphs of the score, where instructions are given for the selection of a target text, the organised extraction of words from this text, and the construction of poems from these extracted words. Cage called these precisely formatted poems constructed out of words extracted according to a specific algorithm "mesostics". In the third and fourth paragraphs the score provides instructions for extracting other words, referring to places and sounds respectively. Because these instructions are intrinsically linked to the remaining instructions of the score, the problem domain is not isolated from the larger task of realizing a performance from the score. Any solution to the instructions in the first four paragraphs must be designed to facilitate the completion of the remaining tasks. The end of the second paragraph of Cage's score clarifies how the preparatory materials of the preceding paragraphs are to be structured into a work to be performed. This is what Cage says:

> Make a tape recording of the recital of the text using speech, song, chant, or Sprechstimme, or a mixture or combination of these. Ascertain its time-length. Subtract that from a total program length, and distribute the thus-determined silence between large parts and chapters of parts and at the beginning and end of the tape. You then have a ruler in the form of a typed or printed text and in the form of a recited text, both of them measurable in terms of space (page and line) and time (minute and second), by means of which the proper position (see 5 below) of sounds (see 3 and 4 below) may be determined.

The problem is then to provide the user with precisely formatted mesostics, where each line is accompanied by an index value (Cages suggested the page and line number), with the words of places and sounds also accompanied by an index value, so that when a recitation of the mesostics is recorded, the textual index values can be mapped onto a temporal index of the recitation recording,

and the recordings of sounds and places can be superimposed onto the recitation recording at the precise moment corresponding to the relationship between index values of the mesostic words, the sound words, and the place words. All this is specified in the 5th paragraph of the socre.

The following subsections provide more detail on the specific problems presented in each of the first four paragraphs of the score. For the reader's convenience, the relevant text from Cage's score is reproduced at the start of each section.

1.1 Selection and formatting of an appropriate target text.

The first paragraph of the score reads: "Choose a book. If it is not in the public domain, obtain permission for its use from those owning the copyright. Failing that, make step 2 in such a way that no relationships of words occur that occur in the original, or encode your text so that the original words are not represented by themselves (change the title of your work accordingly)."

The score constrains the system to use a text which is in the public domain, or for which permission for use has been obtained from copyright holders. Given the resources required for the development and operation of the system, the text must also be written primarily in the English language. Finally the text must also be available or easily convertible into an electronic format that is readable by the system. An enormous range of possible target texts exists. For example, a large sample of literary works is held in the Project Gutenberg repository.[7] All of these meet these criteria: they are out of copyright, and are held in UTF8 text format. Cage's alternative suggestion that a work held in copyright be stealthily encoded to avoid copyright infringement is excluded from this problem domain.

1.2 Choosing a "mesostics row."

The second paragraph of the score begins: "Taking the name of the author and/or the title of the book as their subject (the row), write a series of mesostics beginning on the first page and continuing to the last. Mesostics means a row down the middle…" The specific problem here is to capture, format, and preserve for access by system functions the user's choice of a "mesostic row."

1.3 Emulating the mesostics creation algorithm.

The second paragraph of the score continues: "In this circumstance a mesostic is written by finding the first word in the book that contains the first letter of the row that is not followed in the same word by the second letter of the row. The second letter belongs on the second line and is to be found in the next word that contains it that is not followed in the same word by the third letter of the row. Etc..." The specific problem here is to traverse the mesostic row, and for each letter of the mesostic row traverse the target text to find the next word containing that letter, and traverse the letters of that word

[7] See The Gutenberg Project, www.gutenberg.org.

following the mesostic letter in it to determine whether to save the word, and if saved to move to the next letter of the mesostic row, and finally move to the next word in the target text, until the mesostic row has been traversed, then start again with the first letter of the mesostic row, and repeat the process until the target text has been traversed.

1.4 Separate algorithms for "shorter" and "longer" performances.

The second paragraph of the score continues: "If a shorter rather than longer text is desired, keep an index of the syllables used to represent a given letter. Do not permit for a single appearance of a given letter the repetition of a particular syllable. Distinguish between subsequent appearances of the same letter…" The specific problem here is to perform the same function as explained 1.3 above while including a number of further steps which have the effect of filtering the selection of words from the text, reducing their overall number, so that the recorded recitation of the mesostics becomes "shorter" rather than "longer." There are a number of specific problems here:

1. To create and maintain, for each letter of the mesostic row, including letters which may be repeated, a persistent, searchable, updatable repository of syllables.
2. To break words that are selected by the algorithm explained in section 1.3 into its constituent syllables
3. To identify the first syllable of the words that are selected by the algorithm explained in section 1.3 which contains the letter of the mesostic row by which it was selected.
4. To determine whether this syllable exists in the appropriate repository. If so, move to the next word in the target text; if not, append the word to the repository, save the word, move to the next letter of the mesostic row, and move to the next word in the target text.

1.5 Adding adjacent words to the existing mesostics lines

The second paragraph of the score continues: "Other adjacent words from the original text (before and/or after the middle word, the word including a letter of the row) may be used according to taste, limited, say to forty-three characters to the left and forty-three to the right, providing the appearance of the letters of the row occurs in the way described above. . ." The problem here is to set, or permit the system user to set, an appropriate parameter X (perhaps X = 43, as suggested), to identify in the target text every word selected by the algorithm explained either in section 1.3 or 1.4 above, to identify all whole words preceding that word up to a limit of X characters and spaces from the beginning of the word, to identify all whole words following that word up to a limit of X characters and spaces from the end of the word, to string these together onto a single line, and to preserve that line in an output file. Finally, after the adjacent words have been added the mesostic lines need to be reformatted so that "the appearance of the letters of the row occurs in the way described above. . .", that is, they must appear as "a row down the middle" of the page, a row formatted as uppercase letters.

The problem here, as elsewhere in this project, is to provide a facility to assist the user in following the instructions of the score, not to provide a system which automatically and exhaustively completes those instructions. As the reader may have already discerned, Cage is here giving the user an aesthetic task, to create meaningful poetry by adding adjacent words "according to taste." There are a number of possible software solutions to this problem to be considered, which will be discussed briefly in the following chapter. However, developing a system which chooses words "according to taste" is not one of them. That problem will be solved by the user outside the system.[8]

1.6 Formatting and indexing the mesostics

The last relevant section of the second paragraph of the score is this: "Omit punctuation and capitalize the row, reducing all other capitals to lower case. If at the end of the book or a chapter of it a mesostic is not complete, leave it incomplete or complete it by returning to the first page of the book or chapter and continuing your search for words containing the necessary letters. Having completed the series of mesostics, identify each line by page and line of the original from which it came..." Here Cage provides, perhaps out of the order of an effective system workflow, a scattered list of problems in rendering the mesostics in the prescribed format, dealing with the incomplete traversal of the target texts, and preserving the relative position of all words extracted from the text. The specific problems here are:

1. To identify the letter of a mesostic row in a extracted word and provide functionality to change the case of all letters in the word.
2. To remove all punctuation from extracted words.
3. To provide appropriate functionality to deal with the event where, during the course of the running of the algorithms explained in sections 1.3 and 1.4 above, the end of the target text has been reached before the end of a mesostic row has been reached. Note that Cage distinguishes between "book" and "chapter of it", which presents the user with an important choice regarding the specification of the target text.
4. To preserve the position in the target text of every extracted word with an index value. Cage instructs that this value be a two dimensional array where one element is the page and the other the line number on the page of the word. In the present case, where an electronic target text is used, a more precise and efficient solution is possible.

To help the reader visualize the processes involved, Figure 1 displays how the first mesostic from *The Star Factory*, which became the first page of the recorded recitation for *Owenvarragh*, came to be. A mesostic is developed from a one-word column to a completed poem with adjacent words added "according to taste." Notice that in this case, as was the case with most of the mesostics published by

[8] See the relevant section of the user's manual for the system in Appendix 3.

Cage over the course of his career, taste dictated that only the minimum number of words be added to the column to render the lines evocative and meaningful. Only very rarely did a line reach the length of the parameter suggested by Cage (43 characters to the left and right).

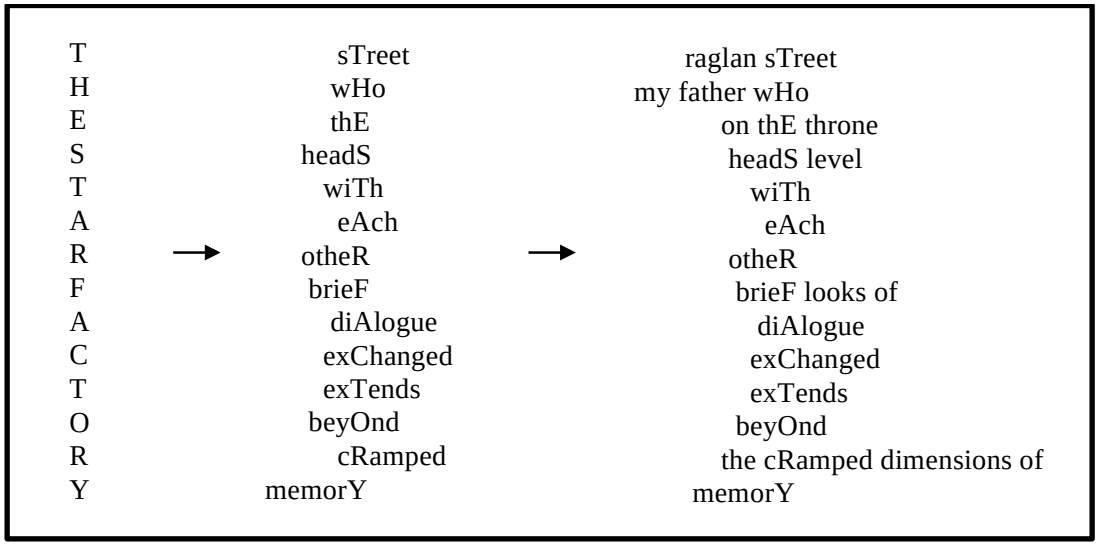


Figure 1. From Mesostic Row to Complete Mesostic[9]

Of course, this is only the first of 137 mesostics extracted from *The Star Factory* and included in *Owenvarragh*. Once the word "memory" was extracted to complete this mesostic, a new page with the column of capitalised letters was opened and the search continued, and this process repeated until the end of the book was reached. The entire text of mesostics is published online, as is a video a performance with Ciaran Carson reciting the mesostics live.[10]

1.7 Identifying places and sounds.

The third paragraph of the score reads: "Make a list of places mentioned in the book and a list for each of the pages and lines where the mention is made for each. If the list once made is unmanageably long, reduce in some chance-determined way, e.g. to a number equal to the number of pages in the book." And the fourth paragraph reads: ""Make a list of sounds mentioned in the book and a list for each of the pages and lines where the mention is made. If the list once made is unmanageably long and baffling because of the large number of kinds of sounds, establish families of sounds and extract from the whole list those related to certain of these."

[9] Figure 1 was creted by Úna Monaghan and published in her Phd. See Monaghan, *New technologies*, p. 63.
[10] Martin Dowling, "Rhythm, Rhymes, and Pleasure: Mesostics on Ciaran Carson's The Star Factory," *Journal of Black Mountain College Studies*, Special John Cage Issue, vol. 4, 2013 (www.blackmountainstudiesjournal.org); for the performance see Martin Dowling's YouTube channel: www.youtube.com/channel/UCzBLSd6k16kPYIATntrnPcA

These instructions pose some serious, but perhaps not insurmountable, problems for a software system. Assuming the problem of traversing a text word by word, which is shared across all the problems already identified, is solvable, the additional problems here are to create lists of words that qualify as a places or sounds, preserving their locational index in the target text. How to provide functionality to identify whether a word in the target text is or is not a “place”, that is, a location on planet earth which can be the subject of a sound recording? How to provide functionality to identify whether a word in the target text is or is not a “sound”, that is, a word which has as its referent a sonic object or experience which can be the subject of a sound recording? Are there ways to use the location of a word in a phrase or sentence to decide whether it is a place or a sound? What heuristics rules of thumb would suggest candidates for inclusion in a pool of words? (For example, a preposition like “in” may likely be followed by a location. Or words located adjacent to certain prepositions may be more likely sounds.) Are there ways to use the format of a word to decide whether it is a place or a sound? (For example, words that are capitalised may be more likely to be places. What databases of words that are places and words that are sounds might exist and be useful resources?) Useful repositories of place words, sound words, and onomatopoeia may exist within online dictionaries and other publicly available databases. The possibility of establishing a means of ‘seeding’ and growing, or scraping and reducing, a repository of words using online resources might provide a significant advantage to a user of the *MesosticMachine*.

Note again that the problem here is to provide a facility to assist the user to follow the instructions of the score, not to provide a system which automatically and exhaustively follows the instructions. The problem specified here is to provide a system which traverses the words of a target text and facilitate the assessment of whether that the word is a sound or a place. The problem is to shorten the task of finding candidate words via eye and the hand. In the end, it may only be possible to identify and exclude all words from the target text that definitely are not places, and definitely are not sounds, giving the user a shorter list. This list will only be as useful as the grammatical or syntactical heuristics employed, and/or only as useful as the databases words to which the words are compared. The development of these heuristics and/or databases are excluded from this problem specification. However, as will be seen in the following chapters, a considerable investment of time, energy, and software skill development has been investing in these areas in the course of the development of this project.

This concludes the specification of the project problems. In the next chapter solutions are proposed and design details relating to those solutions are outlined.

Chapter 2: Proposed Solution and Development Strategy

Following directly on from the problem specification outlined in Chapter 1, this chapter proposes solutions that hopefully are sensible, useful, and demonstrate some flair and originality. The will at least be novel, since the problem domain currently has no known existing software solutions. In the *Owenvarragh* project, digital audio software and hardware were employed where Cage had used audiotape and analog machines,[11] but all engagement with the literary work itself happened via eye and hand. This chapter contuse by outlining a development strategy that follows from these proposed solutions. More refined detail of the system requirements, design, functions, and methods of implementation follow subsequent chapters.

2.1 Proposed Solutions

Taking in turn each of problems specified in sections 1.1 to 1.7 of the previous chapter, this section outlines some proposed solutions.

Formatting a book as a target text. This will be the user's responsibility. The system will recommend to the user that the text selected for realisation as a musical performance be formatted as a UTF 8 text file with a "txt" extension. In order to ensure correct identification of words and syllables, all hyphenation must be removed from the text. Users are also advised that texts held under copyright should not be selected for application unless the copyright holder has given explicit permission.

Choosing the mesostic row. The system shall provide functionality within its graphical user interface to accept from the user in a text field any series of ASCII characters. The system will reformat the received text into a mesostic row which has no whitespaces or punctuation, and save this in a text file for access by system functions.

Software emulation of the mesostic algorithm. The emulation shall read from the mesostic row and target text files, and write to output text files. The emulation shall use nested processes to traverse the mesostic row letter by letter and the target text word by word. In each moment of traversal when a mesostic letter and target text word intersect, the emulation shall employ conditional Boolean logical statements to evaluate the word according to the criteria specified in the score. Boolean outcomes shall govern the invocation of statements which extract words, and which continue the interacting traversals of the letters in the mesostic row and the words in the target text. The emulation shall format and write to a file the extracted words.

[11] This had significant ramifications for the interpretation of the later paragrpahs of the score. See Dowling and Monaghan, "Creating a Circus."

Shorter performances and the syllable repository. There shall be two versions of the algorithm emulation. One, described in the preceding paragraph, creates a longer document of mesostics. Another shall be identical to the first with additional functionality to

- create a file folder accessible to the system containing a set of syllable repositories in the form of a text file for each individual letter of the mesostic row;
- divide candidate words into their constituent syllables by submitting them to the website www.howmanysyllables.com;
- identify the first occurrence of the current mesostic letter in the word, and the syllable where that letter occurs;
- compare that syllable to the list of syllables contained in the appropriate repository file
- if the syllable is not in the file, append it to the file and evaluate the word as one to be extracted, continuing as in the first version.
- if the syllable is in the file, continue the emulation of the algorithm with the word evaluated as not to be extracted.

Adding adjacent words. The system shall employ the functionality already described to traverse the ascii characters of sections of the target text in a forward and backward direction, and use conditional logic to limit the length of traversal according to a set or given parameter, compose an appropriate string of words, and write that string to a file.

Formatting and indexing mesostics. The solutions here are as various as the problems identified: to develop or deploy existing functionality to identify letters of a word and change their case, to remove punctuation and whitespace from target text, to provide a strategy for resolving incomplete mesostic composition when the end of a target text is reached, and to provide a system for giving an index value to all word extracted from the target text. The exact formulation of these solutions will be found within the chosen programming language and development environment. Also, the system will require users to decide whether the target text for the system is the entire book selected, or whether the book should be divided into appropriately sized sections, each in their own text file, such as the existing chapters of the work. In the latter case, the system will run on a sequential series of target texts, with the final products assembled afterward by eye and hand.

Finding Places. The previous chapter suggested two possible solutions to the problems of evaluating whether a word in the target text is a sound or a place: the use of heuristics based on grammatical, syntactical, or formatting characteristics of words and the phrases in which they are located; or the use of databases of sound and place words. Regarding places , the Stanford Natural Language Processing Group provides a tool for the identification of Persons, Organisations, and Places in a target text.[12] Upon consideration and reflection, it is proposed to solve the place problem with a

[12] Stanford Natural Language Processing Group (https://nlp.stanford.edu/).

heuristic, supported by reference to a database of non-places. The decision with regard to place words was informed by the interpretation of the score and the corresponding strategy adopted by Úna Monaghan to collect place words from *The Star Factory* in 2012.[13] An analysis of the words Monaghan extracted from the book shows the potential of a heuristic approach. Because it was decided that candidate words for extraction had to be *actual* places, and because the author Ciaran Carson followed the grammatical convention of capitalizing the first letter of proper nouns, every one of the "place" words extracted from *The Star Factory* begins with a capitalised letter. Functionality that distinguishes words which do and do not begin with a capital letter, and excludes from the former words that are located at the beginning of sentences, will be developed. (There is a risk of missing a place word by this procedure, for example with a sentence like this one: "California is the place you want to be.") This heuristic will generate a significant collection of false positives in the form of proper nouns that are not places, such as personal names, organisations, titles of books and works of art, etc. The system shall also check the candidate words against a database of words that are not places. This database can be tailored for the particular target text, adding the names of the main characters of a novel, for example, or in the case of a book like *The Star Factory* which is about a particular place, excluding words like "Belfast" or "Northern Ireland." This solution provides the user with a lengthy list of candidate places, which the user can then evaluate, removing the proper nouns which are not places.

Finding Sounds. Investigation of this problem produced no useful heuristic approaches to the solution. Sound words appear to escape the grasp of grammatical, syntactical, or formatting heuristics. It is therefore proposed that the system shall traverse the words of the target text (functionality that is shared with every solution outlined here) and compare each word to a database of sounds. A significant amount of time and energy was devoted to researching the possibilities for creating and populating such a database, including thorough research of online sources. The conclusion arrived at through this research was to have the system access a database comprising words taken from two online sources: the online edition of the *Oxford English Dictionary* and the Wordnet database developed at Princeton University. It is possible to scrape from the *Oxford English Dictionary* all words therein having the word "sound" in their definition, which might after aggressive cleaning provide useful seeds of the database. Thus seeded, the WordNet website can be used to gather any further words whose meanings are associated with sounds. The WordNet database organises English language words into tree structures based on hyponymy. A tool called NLTK ("Natural Language Toolkit), which uses the Python programming language, will be used to harvest all of the hyponyms of the word "sound." [14] As with the proposed solution for places, this solution will necessarily generate a significant number of false positive results. Extracted words will have as one of their numerous meanings a sound sense. However this sound sense may not be the sense intended in the particular context in the target text. It is proposed that the system display not only the extracted

[13] Dowling and Monaghan, "Creating a Circus."
[14] The Oxford English Dictionary online: http://www.oed.com/; Wordnet: http://wordnet.princeton.edu/; Natural Language ToolKit: http://www.nltk.org/

words, but the sentence in which they sit, so the user can easily evaluate the sense of the word, and remove it if it is not a sound sense. As with the non-place database, it is proposed that this database be edited and improved with use, and if appropriate tailored to the particular target text.

2.2 Development Strategy

In close consultation with the thesis supervisor, a strategy to develop these proposed solution into a software system was devised in June 2017, following the submission of a Problem Specification. The strategy flows from training received on the MSc in Software Development at Queen's University Belfast. Therefore the solutions here proposed are therefore developed using the Java programming language (version JavaSE-1. 8) in the Eclipse Independent Development Environment (version Neon.3). In addition, extracurricular skill development of skills in the Python languages would be needed to work with the WordNet database. The strategy prioritises those solutions which are the most feasible and which will have the most dramatic impact on the "hand and eye" work presently required by the score, namely the creation of mesostics. These are tackled first, and once completed, solutions in the areas of mesostic finishing and formatting, then sound and place word extraction, are to be addressed.

Some obvious steps in the development of the system into a dissertation submission, such as preparing the requirements specification, the dissertation text, the video documentary, are omitted from this description of the development strategy. The development strategy unfolds according to the following incremental and overlapping steps:

Research on Java data structures and functionality. A thorough revision and deeper exploration of relevant Java functionality is the critical first step of the development strategy. The author began the project by attending the JDKIO conference organised by the Danish JavaGruppen in Copenhagan In June 2017.[15] The research involved the following:

- A review of textbook material on files, streams, and serialization was conducted, paying particular attention to the java IO and NIO packages, and their FileWriter, BufferedReader, and BufferWriter objects.[16]
- A thorough revision of the functionality of the Java Scanner class, its tokens and delimiters, and its methods for traversing a String object (hasNext(), next(), etc) .
- A review of static methods to manipulate String objects (charAt(), indexOf(), substring(), etc.)[17]
- An investigation of StringBuilder object in Java.

[15] See the Development Journal submitted with this dissertation for programme and notes on presentations at this conference. A talk on object oriented design by Yegor Bugayenko was particularly useful.
[16] Paul and Harvey Deitel, *Java: How to Program; Early Objects* (London: Perason, 2015), pp. 644-684.
[17] https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html; https://docs.oracle.com/javase/6/docs/api/java/lang/String.html.

After these preliminary investigations of functionality, it was concluded that the functionality provided for dealing with data structured as lengthy Strings is inferior to that available if data is structured in arrays. Conversion of target text into Arrays of Strings would unlock a wider and more useful palette of functionality. A revision of the Java Collections Framework, particularly the ArrayList, HashMap and TreeMap, followed.

Development of a WordFinder. It is perhaps obvious from this exposition that a core component of the system and therefore a high priority for development would be an object that effectively traverses the words of a target text, and evaluates, extracts, indexes, reformats, and saves words that meet given criteria. This could then be further refined and adapted to the various required criteria (mesostic words, place words, and sound words). This would involve:

- Converting the target text into an array of Strings that are words which are indexed sequentially.
- Providing functionality to appropriately manipulate the case of letters in selected words according to the instructions.
- Providing functionality to preserve the original index value of selected words for use in compilation of a sound file with a sequence of markers (or a text file serving as a proxy for this).
- Providing functionality to place lines of mesostics in appropriate position and format in a persistent file.
- Implementing functionality to read from and write to a file.

Development of a Syllable Searcher. Develop functionality to control the selection of words based on a pool of used syllables will require investigation of and development of facility with coding strategies in Java which make use of Selenium and interfaces with web browsers. Functionality to identify, read from, and write to, a file would accompany this.

Developing a Repository of Sound Words. This involves gaining a basic grasp of Python language syntax, along with the linguistic terms “synset”, “hypernym” and “hyponym,” and developing the following strategy for the construction of a list of sound words:

- Using the “advanced search” tool on the *Oxford English Dictionary* website to download all entries from OED that have the word “sound” in their definition by typing “sound” in the “definition” search field. This produced 3109 entries.
- Clean the OED data, using macros in MS Word, leaving a numbered list of words with their parts of speech. Saved as a tab delimited file.
- Remove duplicate entries, parts of speech columns. The list now has 2795 entries, including duplicate word variants having different suffixes.

- Using NLTK in a Python shell to access the WordNet databse, produce a list of synsets of the word “sound” (there are 24 such sysnsets), along with their definitions (see Appendix 4 for some of the routines eventually used to query the WordNet database for).
- Split the list into two, distinguishing between (a) synsets of meanings relation to auditory sensation, (b) synsets of non-auditory meanings (body of water, logical validity, health, etc)
- Remove all words from the non-auditory synsets from the OED list.
- Add all words from the auditory synsets to the OED list, if they are not already included
- Add words from the list of sounds taken from The Star Factory in the Owenvarragh project to the OED list if they are not already included.
- Eliminate duplicative suffixes from the list, leaving only words or word fragments that identify words in the list with a “begins with” search in Java. .
- Test the file repeatedly with the MesosticMachine SoundGetter object and evaluate the list for unnecessary false positives. Remove words if necessary .

Development of GUI and User Guide. This involves research into the available tools for developing a GUI for a Java system. It was decided to stick with the WindowBuilder functionality resideing “in hous” in the Eclipse IDE. This provided a straightforward classic Windows look and feel appropriate to the project, and was easy to construct.

2.3 Design Details

This description of proposed solutions and the development strategy brings justifies thinking of the developed system as a tripartite structure aligned with three areas of functionality, all housed within a GUI which permits users to configure the parameters of the system:

(1) A mesostic making subsystem which deals with a target text and a preset mesostic row,
(2) A sound and place getting subsystem which deals with a target text and comparator repository text files
(3) A mesostic finishing subsystem which deals with mesostic files output by subsystem (1) to create usefully formatted text files.

A more detailed illustration of these design details follows in chapter 4.

Chapter 3: Requirements analysis and specification

This chapter contains three sections. The first provides reflective material from previous experience implementing the functions of the proposed system "by eye and hand", including an interview with a previous collaborator. The unique situation of analysing and specifying requirements from a musical score are highlighted. This is followed by a section which analysis and illustrates a range of anticipated use cases for the system and the system outputs envisioned for those use cases. The final section summarises the requirements specification which is presented in full in Appendix 2.

3.1 Elicitation of Requirements

Often an analysis of requirements will include an account of how they were elicited from clients or potential users of the system under development. The elicitation process can take many forms and use a variety of methodologies. This scenario involves the interpretation of a published musical score by a deceased composer. The requirements are prescribed by the score, not elicited from it. The software developer therefore has the same artistic obligation to these prescriptions as do conductors and musicians who set out to perform from a composer's score. They must interpret the score in a way that adheres truthfully to the composer's intentions. Within the bounds of that adherence, the artist/producer is free to bring their own skill, originality, and flair into play.[18] The author is one of the few people living to have direct experience collaborating on an implementation John Cage's score by eye and hand. Another one of those few people, Úna Monaghan, co-creator and producer of the *Owenvarragh* performances, kindly met with me for an elicitation interview in June 2017. The following paragraphs draw on that interview and the publication which documents the interpretation of the Cage score guiding the development of *Owenvarragh* to highlight some of the design problems that bear on the specification of system requirements.

One design problem is dealing with what the system should do when reaching the end of a file. The score provides two options: "If at the end of the book or a chapter of it a mesostic is not complete, leave it incomplete or complete it by returning to the first page of the book or chapter and continuing your search for words containing the necessary letters." In the Dowling/Monaghan implementation, "where a mesostic was completed on the last or penultimate page of a chapter, a new one was not started in that chapter, but with the title of the following chapter."[19] The *MesosticMachine* will not behave this way, because "page number" is not a relevant variable. This suggests that a requirement of the system would be to establish an index value X that approximates the beginning of the penultimate page and run the mesostic "while X < chapter.length() – X)." This would possibly address the additional requirement to avoid runtime errors that are thrown when the end of a file is reached. Also, the system could be further developed to handle Cage's alternative prescription to return to the beginning of the chapter. To address this use case, an alternative **if** statement would be required to

[18] A footnote on the ethics and aesthetics of interpreting a score.
[19] Dowling and Monaghan, "Making a Circus", p. 9

complete the mesostic that is incomplete when it reaches the end of the file by returning to the first word in the chapter, then terminating the programme when the last mesostic line is completed, could easily be added to the system.

Another design problem arises from Cage's use of the word "Etc." He neglected to specifically prescribe what happens when the last letter of the mesostic row is reached. Here is the relevant passage: "The second letter belongs on the second line and is to be found in the next word that contains it that is not followed in the same word by the third letter of the row. Etc." The Dowling/Monaghan interpretation offered a novel solution to this problem, which was justified by the particular form the mesostic row took in *Owenvarrragh*. "The relative rarity of the letter Y in the English language, and its frequent occurrence at the end of words, combined with our decision to adopt the index of syllables constraint (see below), helped us to decide to select the next word that contains the last letter of the mesostic without worrying about what letters follow it in the same word."[20] To deal with the possible occurrence of other letters of the alphabet that are less rarely used, the *MesosticMachine* uses the first mesostic letter as the comparator for the last.

Another relevant conclusion of the Dowling/Monaghan interpretation is that it is likely that the vast preponderance of users will wish to use the syllable index, because a longer text would produce an excessively lengthy performance. One of the great advantages of the functioning *MesosticMachine* would be to allow a long and short version to be produced quickly. This would meet a the need to compare alternative strategies for length and artistic potential. The hand and eye approach does not allow the user to predict with any certainty what length would result, until a significant amount of time and energy has already been committed to one or the other strategy.

With regard to the collection of mesostics that are used in a performance, the system must produce outputs for two scenarios: One set of mesostics must have adjacent words added "according to taste" and another must not. Cage allows the artist to stop after having made mesostics with one word per line, with punctuation removed and all letters except the central column in lower case. It is unlikely that this will be a popular user scenario. Dowling and Monaghan concluded:

> In our judgement, more than a single word for each line of the mesostics was required to make manifest the rich poetry in Ciaran's prose. At the same time, what appeared most beautiful, elegant, and fitting to us were mesostics produced by using as few additional words as possible. The mesostics have an aesthetic value and meaning that is independent of the prose out of which they are lifted. But because of the "bunching" effect of the procedure described above, many of the mesostics contain words that are located very close to each other in the book.[21]

[20] Dowling and Monaghan, "Making a Circus", p. 11.
[21] Dowling and Monaghan, "Making a Circus", p. 11.

Dowling and Monaghan also came to useful conclusions about the appropriate limit on the number of adjacent words that should be allowed to be added:

> We allowed the font "perpetua" size 16 and the left and right margins on an A4 page of 2.54 centimetres to limit the number of characters to the left and the right. This was always less than 43 characters, so it appeared to be in keeping with Cage's instruction. We used only whole words that fitted. This instruction might be read to imply that the word containing the next letter of the row be allowed as part of the added words, if it were to fall within the limit. However such repetitive use of words in a mesostic fit neither our taste nor, apparently, Cage's own in the *Finnegan's Wake* realisation. Accordingly we have interpreted this instruction to have a further clause, "and providing that the word from the adjacent lines of the mesostic is not used." In the end, the space limit was rarely reached before a pleasing mesostic had been made.[22]

Cage anticipated that a chosen text might contain an excessive number of places and sounds. The relevant passage reads: "If the list once made is unmanageably long, reduce in some chance-determined way, e.g. to a number equal to the number of pages in the book." (The reader will remember that his chosen text was James Joyce's Finnigan's Wake). The Star Factory was another such example. Monaghan counted a total of 1,052 places mentioned in the book, which definitely qualifies as an "unmanageably long" list. How to reduce the number of places to one page per chapter? The system must translate from the word index to page location. Monaghan followed Cage's recommendation to reduce the number of places to the same number of pages as in the book, 291, and adopted a dice-throwing procedure and other strategies of indeterminacy to select from them"[23] On the other hand, the list of sounds taken from *The Star Factory* was not unmanageably long, so this scenario did not arise for sounds.

The Dowling/Monaghan interpretation also raises some interesting complications for interpreting Cage's instructions to collect place words. The use of the "uppercase letter" heuristic will not remove the uncertainty about the actuality of some of the places mentioned, because there are real places that are not mentioned by proper name in The Star Factory. For example: "'the biggest shipyard in the world' was included because we identified it as Harland and Wolff, but in the same paragraph 'the biggest linen spinning mill' and 'the longest rope walk' were not included (p. 179)," but might have been researched and identified.

In an interview conducted on 23 June 2017, I asked what a system coud do to speed up the work of the sound engineer tasked with paragraphs 5 through 7 of the score and she suggested an indexed

[22] Dowling and Monaghan, "Making a Circus", p. 11.
[23] Dowling and Monaghan, "Making a Circus", p. 16.

sound file which marked in exact temporal sequence all of the components of the “tape part”: each mesotic line, each sound word, and each place word.

3.2 Analysis of Use Cases

A project to produce a musical realisation of a literary work will ideally be taken up by a team, each with their own use cases and requirements, though the team may only be one or two people who take on several roles and are several actors at different phases of the project’s development. These actors include:

- A Mesostics Creator/Producer who will require the system to produce an accurate set of mesostics from a target text and a set mesostic row.
- A Sound Engineer who will require the system to produce accurately indexed documents for the mesostic words, sounds, and places, all essential components for field recording and construction of the “tape” part as prescribed by the score
- A System Administrator, namely the author and main user of the system in its initial phase of operation, who will require a variety of data to monitor system outputs and refine input parameters to improve system functionality.
- Performers who will require the system to produce reference documents for the mesostic recitation that forms the skeleton and timeline of the performed work:
- A Publisher who will require a clean and attractive version of the mesostics for publiscation online or in print.

Figure 1 displays the various actors engaged in different but interrelated ways with the score. The next sections explores what type of outputs these various actors might require from the system.

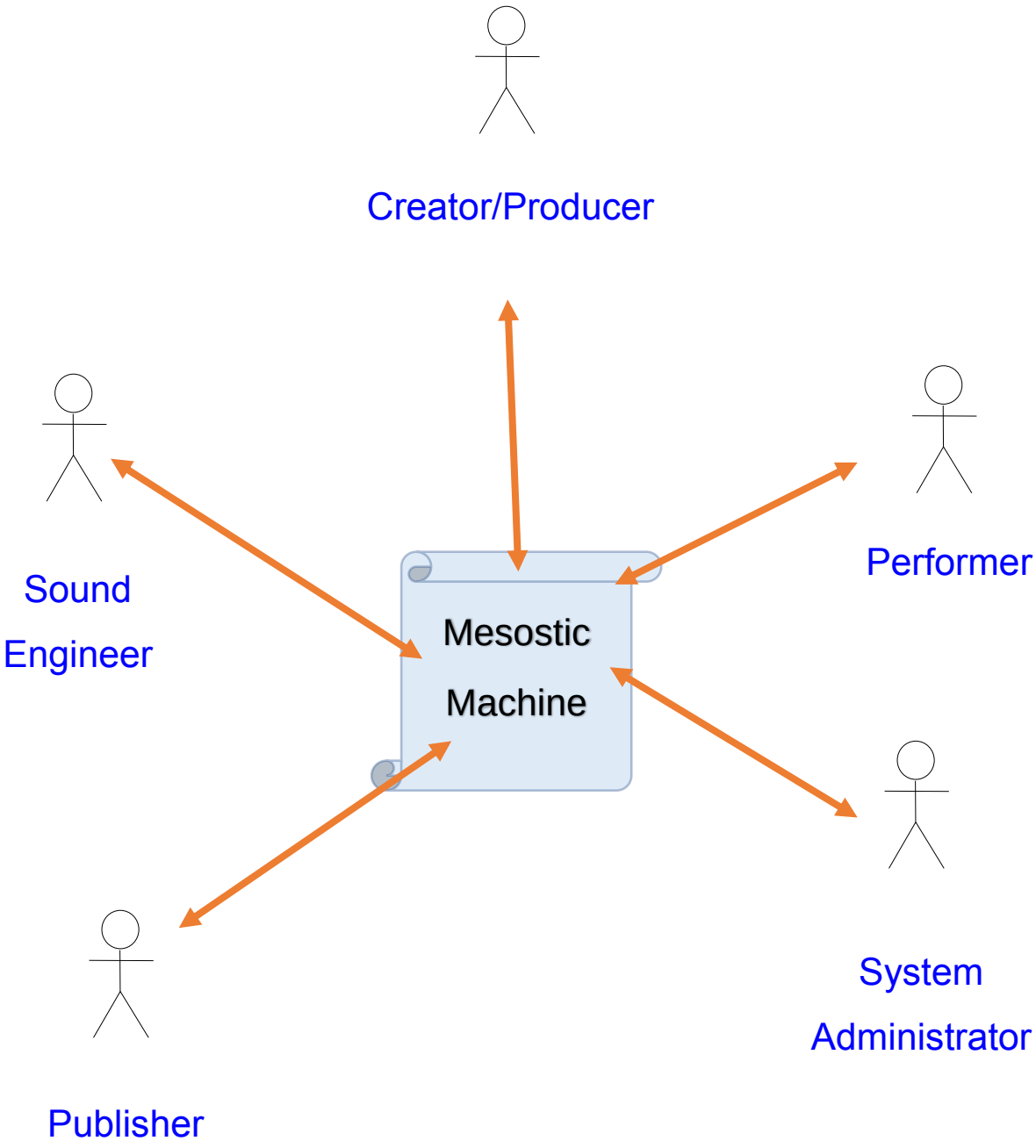


Figure 2: System Users

3.3 Requirements Specification and list of system functions

The purpose of this section is to set out detailed descriptions of how the system services various use cases, and to indicate how the requirements can be tested. Figure 3 below shows how particular outputs are related to the scenarios confronting different users. The relevant sections of Cage's score have been recast as a set of software system requirements from the perspective of its potential users. The result is placed in Appendix 2. These requirements are divided into four sections: (1) choosing a book, (2) making mesostics, (3) getting places, and (4) getting sounds.

The first section defines the data model for the system, and these requirements comprise the preconditions and inputs for the remaining system requirements. These requirements lie outside the system; it is the system user's responsibility to ensure that these requirements are met so that the system can run smoothly. Once the requirements of this secton are met, the GUI can receive information about the location of relevant input documents and the system will be ready to function.

The second section contains the following functions, all of which have a properly formatted and located target text and mesostic row as preconditions and inputs, and newly created text files as outputs:

- Save a mesostic row to a text file
- Provide a software emulation of the algorithm for creating a mesostic
- Create and save a full list of completed mesostics to a text file (with and without a syllable filter)

This section also contains the following functions all of which have the output files of the previous functions as preconditions and inputs, and newly created text files as outputs:

- Add adjacent words to lines of a mesostic text file
- Format the lines of a mesostic text file
- Format the final incomplete mesostic of a mesostic text file
- Index the lines of a mesostic text file

The third and fourth sections contain the following functions, and have as their preconditions the creation of suitable heuristics and comparator repository files.

- Find and save in a text file words that are places
- Index the lines of a places text file
- Reduce through chance-determined selection of lines the size of a places text file
- Find and save in a text file words that are sounds
- Index the lines of a sounds text file

- Reduce through chance-determined selection of lines the size of a sounds text file

.

In the following chapters show how these functions are integrated into the system design and implementation, and a testing strategy was employed throughout the development of the system.

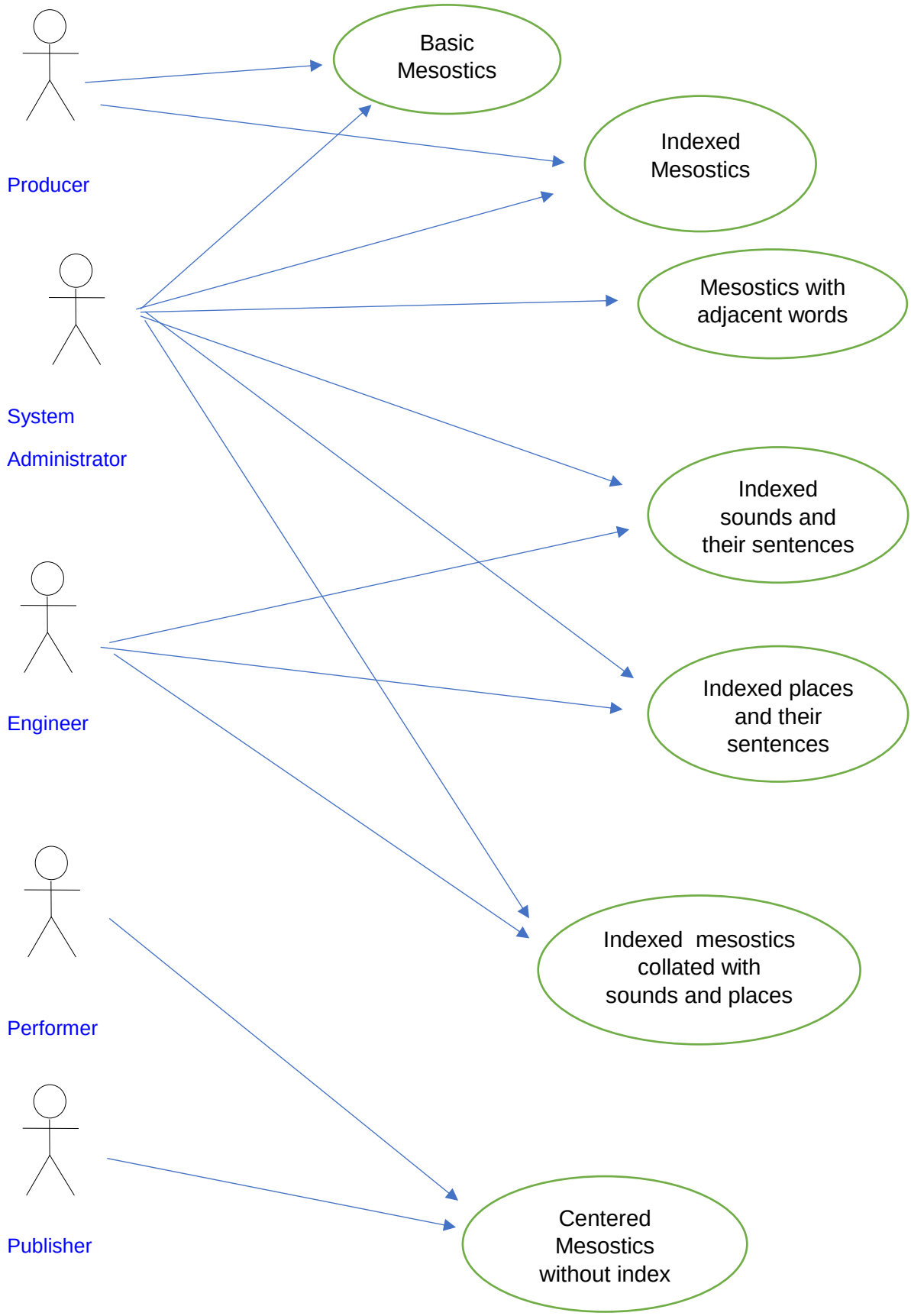


Figure 3: Use Cases with Required Outputs

Chapter 4: Design Illustration

This chapter illustrates the design of the system, shows where the functions addressing specific requirements are located in Java classes, where those classes are imbedded in the GUI, and the recommended user paths through the GUI. A more detailed and less abstract exposition of the system and its main objects, which drills down to the actual code, follows in the next chapter.

The system has four main parts, or subsystems: mesostic making, sound and place getting, mesostic finishing, plus a the graphical user interface which incorporates user input of precondition information gives the user control over the functionality of the first three parts. Each of the three main parts of the system are organised under an interface: the mesostic-making objects in the system implement the ImesosticMaker interface; the sound and place getting objects implement the Igetter interface.; and the mesostic-finishing objects implement the ImesosticFinisher interface. The design does not exploit the full capabilities of the strategy of "programming to an interface", but adoption of this strategy provides llustrative clarity and reduces the proliferation of method names. The objects that implement ImesosticMaker and ImesosticFinisher have a single method, and the two objects implementing Igetter only two.

The system is object-oriented in its design. The mesostic-making subsystem includes three core, ground-level objects that are integrated into higher-level interface-implementing objects do the work of create files housing series of mesostics. The place- and sound- getting objects reside in the second subsystrem sitting alongside the mesostic making section. Because it takes as inputs the output of the mesostic-making section, the mesostic finishing section can be viewed as lying beneath, or anterior to, the mesostic making section.

These three main system parts are illustrated in the UML class diagrams in Figures 4, 5, and 6 on the following pages. These figures show the private variable attributes of each object, their overridden methods, their relationships to each other, the input text files they receive and the output text files they produce.

Figures 7, 8, 9 and 10 are sequence diagrams depicting the recommended user paths through the GUI. The user begins with a setup routine (Figure 7, red arrows), then proceeds to a mesostic-making routine (Figure 8, purple arrows), then a mesostic-finishing routine (Figure 9, blue arrows), then a place and sound getting routine (Figure 10, pink arrows)

Figures 10 and 11 are a hybrid UML/Sequence diagrams showing where the main functioning objects of the system lie within the graphical user interface. In the following chapter, salient details of the code in the classes depicted here are unpacked and explained.

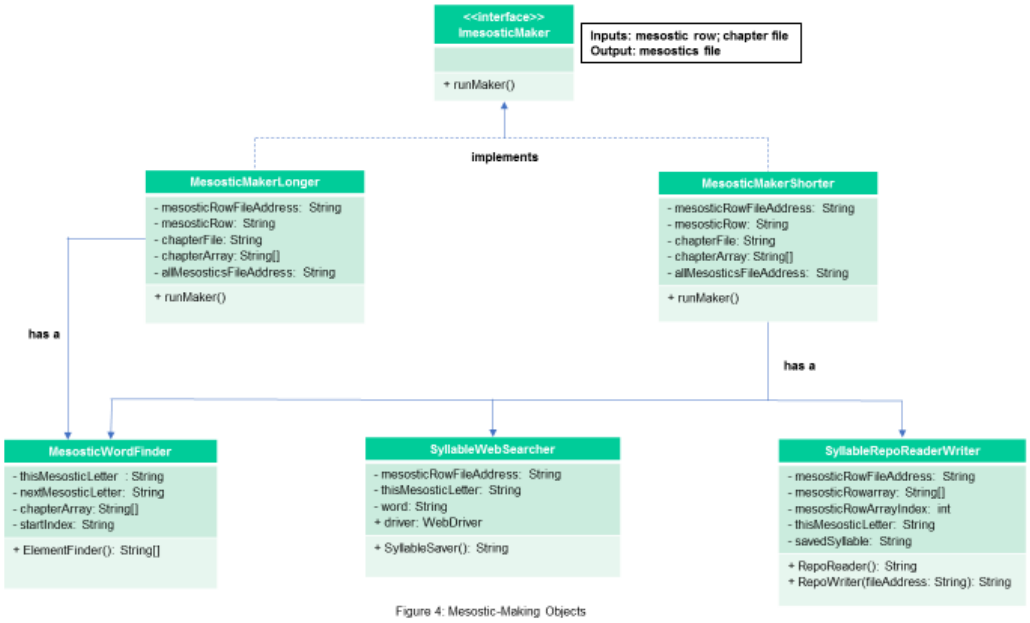


Figure 4: Mesostic-Making Objects

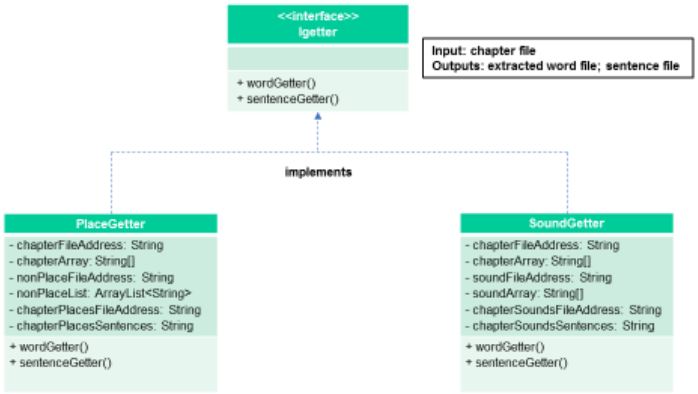


Figure 5: Place and Sound Getting Objects

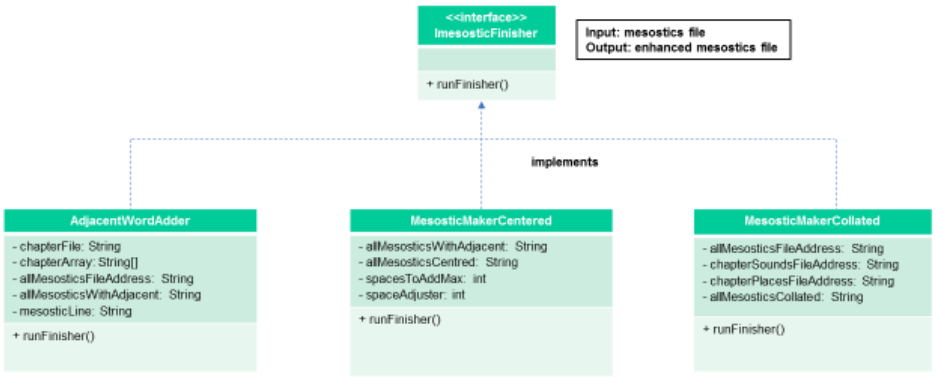
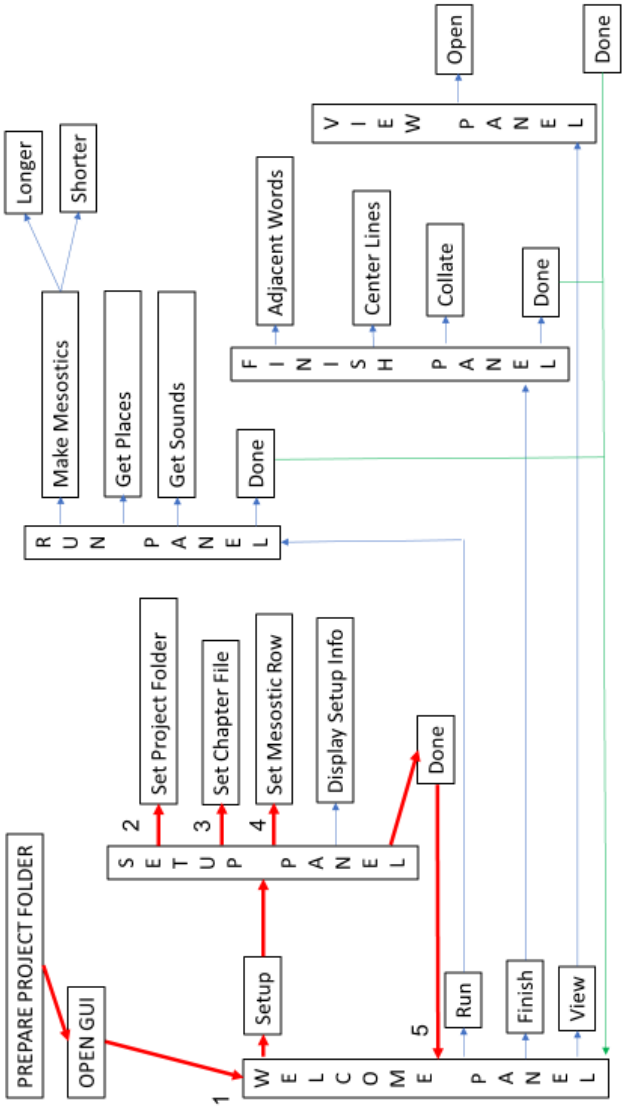
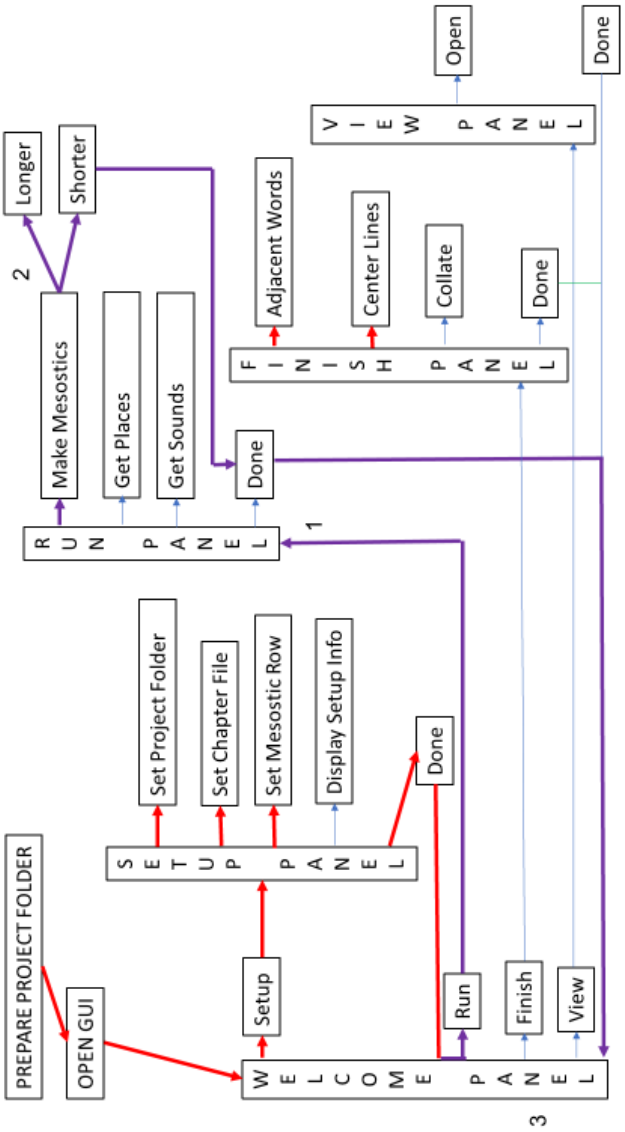
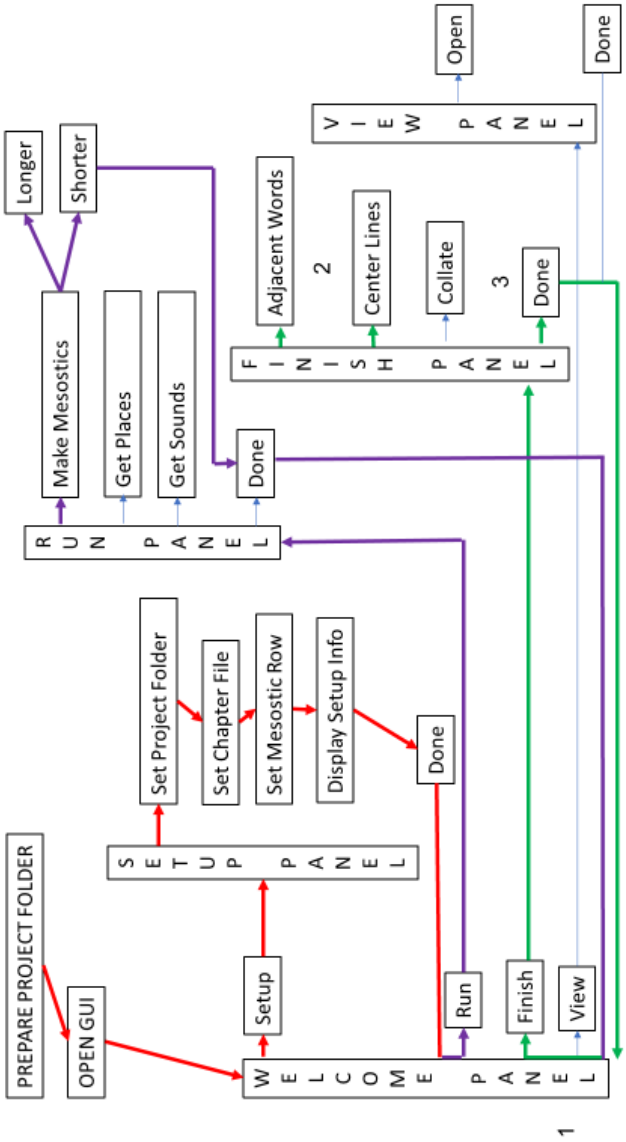
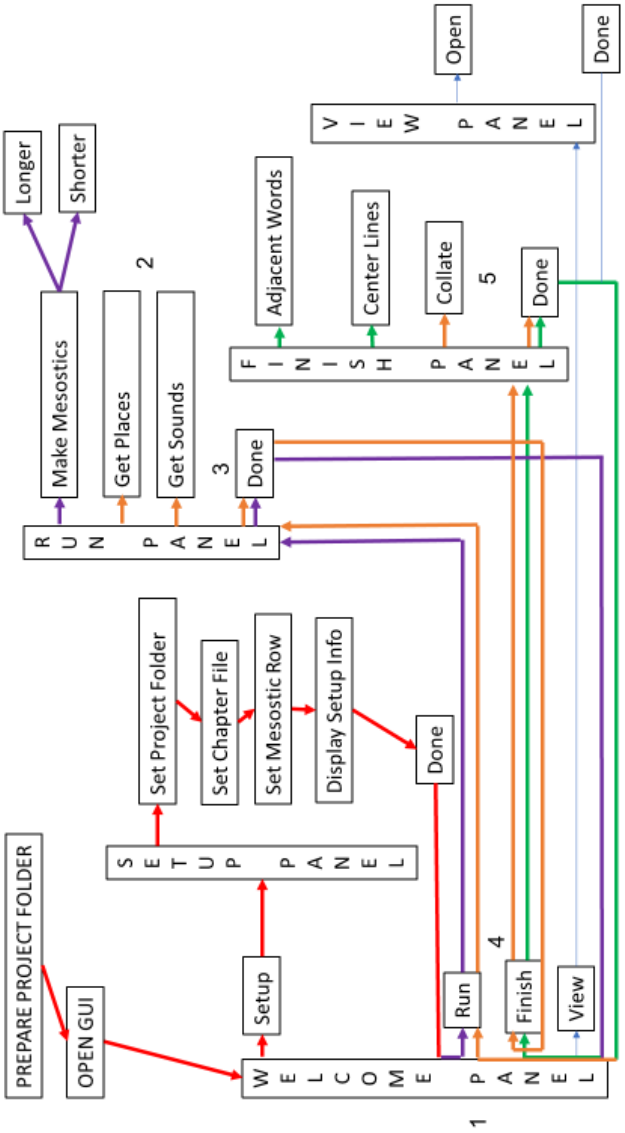


Figure 6: Mesostic Finishing Objects


PREPARE PROJECT FOLDER
OPEN GUI
1
WELCOME PANEL
Setup
2
SETUP PANEL
Set Project Folder
3
Set Chapter File
4
Set Mesostic Row
Display Setup Info
Done
5
Run
Finish
View
RUN PANEL
Make Mesostics
Longer
Shorter
Get Places
Get Sounds
Done
FINISH PANEL
Adjacent Words
Center Lines
Collate
Done
VIEW PANEL
Open
Done


PREPARE PROJECT FOLDER
OPEN GUI
WELCOME PANEL
Setup
Run
Finish
View
SETUP PANEL
Set Project Folder
Set Chapter File
Set Mesostic Row
Display Setup Info
Done
RUN PANEL
Make Mesostics
Get Places
Get Sounds
Done
Longer
Shorter
2
1
FINISH PANEL
Adjacent Words
Center Lines
Collate
Done
VIEW PANEL
Open
Done
3


PREPARE PROJECT FOLDER
OPEN GUI
WELCOME PANEL
Setup
Run
Finish
View
1
SETUP PANEL
Set Project Folder
Set Chapter File
Set Mesostic Row
Display Setup Info
Done
RUN PANEL
Make Mesostics
Get Places
Get Sounds
Done
Longer
Shorter
FINISH PANEL
2
Adjacent Words
Center Lines
Collate
3
Done
VIEW PANEL
Open
Done


PREPARE PROJECT FOLDER
OPEN GUI
WELCOME PANEL
Setup
Run
Finish
View
1
SETUP PANEL
Set Project Folder
Set Chapter File
Set Mesostic Row
Display Setup Info
Done
4
RUN PANEL
Make Mesostics
Get Places
Get Sounds
Done
2
3
FINISH PANEL
Adjacent Words
Center Lines
Collate
Done
5
VIEW PANEL
Open
Done
Longer
Shorter

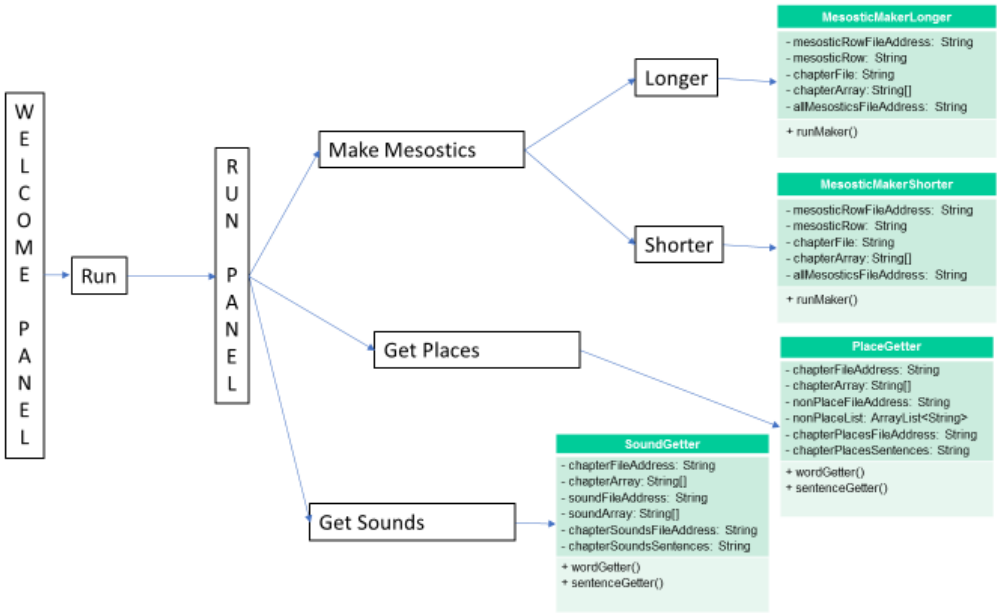


Figure 11: System Objects Encapsulated in the Run Panel

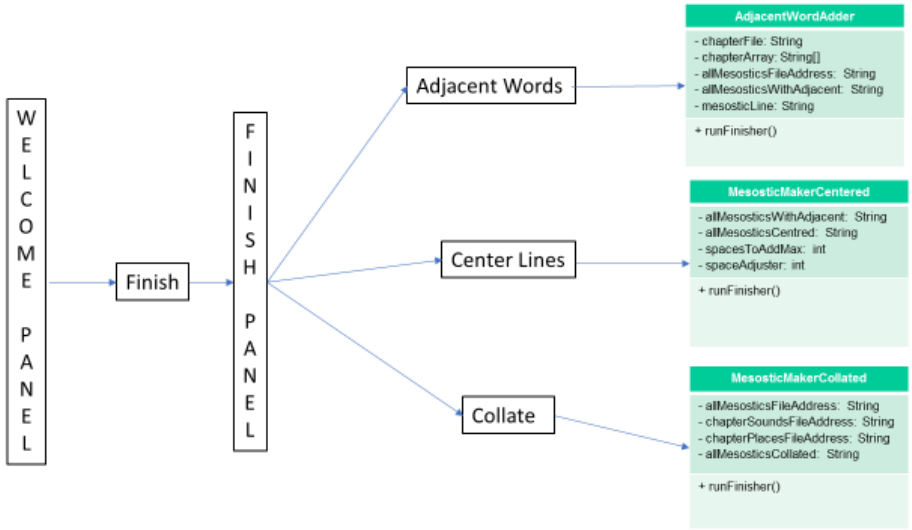


Figure 12: System Objects Encapsulated in the Finish Panel

Chapter 5 : Implementation

5.1 Methodology

The methodology of this project is highly influenced by the MSc in Software Development at QUB curricular emphasis on the Java programming language and its techniques, Object Oriented Design philosophy and SOLID design principles also guide the implementation. Independent research on Java design also inform the methodology. The writings, webinars, and presentations by Yegor Bugayenko, who might be considered an object oriented design purist unhappy with the "procedural" aspects which remain embedded in the Java environment, were a motivation.[24] The system is envisioned as a set of anthropomorphic objects, with their own attributes and behaviours, which interact in loosely coupled autonomy from each other. Static methods, "getter" methods, and "setter" methods are avoided. The goal is to provide a software solution whose functionality is transparently understandable and which uses as few lines of code as possible. Ideally objects have only one or two methods that return one simple object for other objects to use. The only exception to this in the present software system is the GUI which, because it was developed using the WindowBuilder framework in Java, has an ungainly 700+ lines of code and employs various prefabricated static methods. All the other objects in the system are smaller, more elegant, and therefore, it is hoped clearly understandable to other developers.

Ideally if the code is clearly annotated and designed according to these principles, the remainder of this chapter would superfluous to future users and developers. Indeed, the exposition in the following section very closely replicates the structure and the annotations of the actual classes residing in the source folder of the system which are the tools for object creation in the GUI. The following section walls through the system, and the chapter concludes with some commentary on the testing strategy employed in this implementation.

5.2 Specification of Main System Components

This section specifies more precisely some significant aspects of the implementation, including examples of notable sections of code. Each object in the system is discussed, starting with the ground level objects holding the core functions of the system.

MesosticWordFinder. This is perhaps the most fundamental object in the system, and the first to be developed. It is an array-traversing and word-finding object which returns a two-element string array containing an extracted word preceded by its index value formatted as a string. The objects takes as

[24]. Yegor Bugeynko blog http://www.yegor256.com/. For example: "Getters/Setters. Evil. Period", http://www.yegor256.com/2014/09/16/getters-and-setters-are-evil.html ; "Who is an Object?": http://www.yegor256.com/2016/07/14/who-is-object.html.

input variables a string representing the path address of the mesostic row file and the target chapter file converted into an array of strings. Because it is reading from target files in the project folder, the object makes use of the IOException, Files, and Paths packages of the java.io and java.nio libraries. Because the wordfinding algorithm requires the input of both a given mesostic letter and the letter following in the row, string variables for successive pairs of letters in the mesostic row are also declared. Inside this object's constructor, letters in the mesostic row are instantiated as successive pairs by taking the constructor's letter as "this" letter and adding to the index to get the "next" letter. If "this" letter is the last in the row, it is paired with the first letter of the mesostic. Here is the code snippet from the constructor:

```
this.thisMesosticLetter = mesosticRowArray[mesosticRowArrayIndex];
if (mesosticRowArrayIndex + 1 < mesosticRow.length()) {
this.nextMesosticLetter = mesosticRowArray[mesosticRowArrayIndex + 1];}
else this.nextMesosticLetter = mesosticRowArray[0];
```

A starting index to identify where in the chapter file to begin an iteration of the method is also declared. This variable will in normal usage be initialised at "0".

The object has a single method, ElementFinder(). For each letter of the mesostic row, this method traverses the array of words in the chapter and returns the appropriate word preceded by the value of the array element of that word. Here are the steps, with some code snippets, that make this happen:

(1) Convert string index variables to integers:

```
Integer startIndexAsInteger = Integer.parseInt(startIndex);
int wordIndex = startIndexAsInteger.intValue();
String word = chapterArray[wordIndex];
```

(2) Initialise a substring for dividing the target word:

```
int letterIndex = 0;
String wordSegment = word.substring(letterIndex);
```

(3) Inside a **for** loop traversing the chapter array, find the next word, starting from the start Index location, that has the first letter;

```
if (chapterArray[i].toLowerCase().contains(thisMesosticLetter)){
        word = chapterArray[i].toLowerCase().replaceAll("\\W", "").trim();}
```

(4) splits the word at that letter into two strings:

```
for (int j = 0; j < word.length(); j++) {
        letter = String.valueOf(word.charAt(j));
        if (letter.equals(thisMesosticLetter)) {
                letterIndex = word.indexOf(letter);
                wordSegment = word.substring(letterIndex);
                break;}}
```

(5) Test whether the second letter is in the second string;

```
if (!wordSegment.contains(nextMesosticLetter)) {
        letter = letter.toUpperCase();
        word = word.substring(0, letterIndex) + letter +
word.substring(letterIndex + 1);
        break; } else wordIndex = wordIndex + 1;
```

(6) Place steps (2) to (4) in a **for** loop that moves to the next word if (4) is true, or formats and saves the word if (4) is false, then moves to the next word

(7) Returns String[] output = { wordIndexAsString, word }, the saved and formatted word preceded by the index of its element in the chapter array.

SyllableWebSearcher. Having developed an object that traverses the target text, extracts and formats appropriate words, and appends them to a new line of a mesostics file, it was time to develop the syllable filtering functionality to produce a “shorter” rather than “longer” version of the mesostics.. *SyllableWebSearcher* is the first of two objects designed to tackle this problem. The object does the following:

(1) takes a word from target text as parameter
(2) conducts a website search to find the syllabic division of the word
(3) identifies and returns the appropriate syllable
(4) if the target website does not provide a syllable division, saves the whole word and prints a message

Because this object makes use of a publicly available website which divides words into syllables, it uses WebDriver and WebElement objects available from openqa.selenium library. As with the

previous object, it makes use of the IOException, Files, and Paths functions of the java.io and java.nio libraries, and it declares private variables which represent the path address of the mesostic row file, mesostic row letter, and target chapter word. These are set in the constructor, along with a FireFox WebDriver. *SyllableWebSearcher.*has one method SyllableSaver(), which gets the www.howmanysyllables.com/ website with the WebDriver and retrieves the target "word" variable divided into syllables from the website.

```
WebElement input = driver.findElement(By.name("SearchQuery_FromUser"));
input.sendKeys(word);
WebElement submit = driver.findElement(By.id("SearchDictionary_Button"));
        submit.click();
```

Unfortunately on some pages of the website there is no syllable division given, which causes selenium to throw a NoSuchElementException. If this exception is thrown, the word will be saved "as is" and the user must edit the repository manually to record the correct syllable used. The following code accomplishes this:

```
WebElement result;
String[] wordAsSyllables = new String[1];
try {
    result = driver.findElement(By.id("SyllableContentContainer")).find
    Element(By.className("Answer_Red"));
    wordAsSyllables = result.getText().split("-");
    }
catch (Exception ex) {
    System.out.println("Syllable division not found. Saving whole word as
    one syllable");
    wordAsSyllables[0] = word;}
```

Finally, the method returns the saved syllable as a string variable.

SyllableRepoReaderWriter. Once *SyllableWebSearcher* does its job and returns a properly formatted saved syllable, the next task is to develop an object which will read from the appropriate syllable repository files which correspond to each mesostic letter, and if appropriate append the saved syllable to that file. *SyllableRepoReaderWriter* accomplishes the necessary tasks by using the BufferedReader() and BufferedWriter() methods available in Java libraries, declaring the usual private variables for the mesostic row and the letters in the row, and taking the saved syllable from the

SyllableWebSearcher into its constructor. This object has two methods, RepoReader() and RepoWriter(). RepoReader() has to engage with a collection of repository files, one for each letter in the mesostic row, then find and return the correct file. It manages this by creating a HashMap of syllable repository file addresses and instantiating and returning the appropriate file path for each mesostic letter: :

```
Map<Integer, String> syllableRepositoryDir = new HashMap<>();
for (int i = 0; i < mesosticRowArray.length; i++) {
        String mesosticLetterFileAddress =
        "C:\\Users\\Martin\\Documents\\MSc Software\\Final Project\\Star
        Factory Text\\Syllable Repositories\\MesosticLetter"
                + (i+1) + ".txt";
        syllableRepositoryDir.put(i, mesosticLetterFileAddress);                }
```

How does the RepoReader choose the appropriate repository file? The user needs to set these files up properly as part of the setup routine.[25] The first line of each repository file contains the mesostic letter preceded by the index value of that letter in the mesostic row. These two values are matched to the corresponding values set in the constructor to identify and return the appropriate repository. The following code accomplishes this:

```
for (Integer key : syllableRepositoryDir.keySet()) {
        mesosticLetterRepoFileAddress = syllableRepositoryDir.get(key);
        BufferedReader br = new BufferedReader(new FileReader(new
        File(mesosticLetterRepoFileAddress)));
        String line = br.readLine();
        int n = mesosticRowArrayIndex + 1;
        String m = n + thisMesosticLetter;
        if (line.startsWith(m)) {
        break;}br.close();
```

The method then returns the appropriate address for the target letter. The second method, RepoWriter(), takes this address from the RepoReader() so that it can read the correct repository file. It instantiates a Boolean variable to test whether saved syllable from *SyllableWebSearcher* is in the repository. RepoReader() returns the value of this Boolean. If the Boolean is false, the method appends the syllable to the appropriate repository. Here is the code:

[25] See the *MesosticMachine User's Manual* for details of project folder requirements, Appendix 3.

```
BufferedReader br = new BufferedReader(new FileReader(new
File(fileAddress)));
BufferedWriter bw = new BufferedWriter(new FileWriter(new File(fileAddress,
true));
boolean RepositoryHas = true;
for (String line = br.readLine(); line != null; line = br.readLine()) {
        if (line.equals(savedSyllable)) {
                System.out.println("Syllable is in Repository. Start again with
                next word.");
                RepositoryHas = true;
                break;
                } else RepositoryHas = false;
        } br.close();
if (!RepositoryHas) {
        System.out.println("Syllable not Repository.");
        bw.newLine();
        bw.write(savedSyllable);
        System.out.println("Syllable written to Repository.");
} bw.close();
return RepositoryHas;}
```

MesosticMakerLonger and *MesosticMakerShorter* are similar objects that encapsulate the previous objects in a method that traverses the entire target text within a **while** loop, within which lies a **for** traversing the mesostic row. These objects repeat the behaviours of the encapsulated methods, creating mesostics one at a time until the end of the target text file has been reached. The **for** loop in these objects is preceded and followed by statements that write a buffering opening line with index 0 and a buffering closing line with an index of 10000, which was found necessary for the smooth operation of the finishing objects. *MesosticMakerLonger* instantiates *MesosticWordFinder* and appends the output of its ElementFinder() method to a new file in the project directory: allMesostics.txt. With this method, the system is finally creating a useful output. *MesosticMakerShorter* has the same structure, but brings all the syllable filtering methods into play in the appropriate order. First it instantiates *MesosticWordFinder* and assigns the output of its ElementFinder() method to a variable which is passed to the instantiated *SyllableWebSearcher*, assigning the output of its SyllableSaver() method to a variable which is in turn passed to the instantiated *SyllableRepoReaderWriter*. Finally, if the saved syllable is found to be already in its

repository, the method starts again with the same mesostic letter, or else it appends a new line to the allMesostics.txt file and advances the variables to the next mesostic letter and the next word in the chapter. Here is the code for this final step within the nested **while**/**for** structure:

```
if (srw.RepoWriter(targetRepository)) {    i--; } else {
    BufferedWriter bw = new BufferedWriter(new FileWriter(new
    File(allMesosticsFileAddress), true));
    bw.newLine(); bw.write(mesosticLine); bw.close(); }
integer = new Integer(result[0]) + 1;
startIndex = integer.toString();
```

PlaceGetter traverses the target chapter and extracts words that begin with an uppercase letter that are not located at the beginning of a sentence, filtering these through a repository of words that are not places. Its set of private variables and constructor are similar to the other objects which access files in the project folders, and so is its use of typical Java libraries. *PlaceGetter* has two methods, WordGetter() and SentenceGetter(). WordGetter() functions as follows:

(1) It uses a TreeMap to temporarily hold all the words in the chapter that begin with an uppercase letter, but are not one word ahead of a word that has a full stop:

```
Map<Integer, String> placeWordMap = new TreeMap<>();
for (int i = 1; i < chapterArray.length; i++) {
        if (Character.isUpperCase(chapterArray[i].charAt(0)) &&
        !chapterArray[i - 1].contains(".")) {
        placeWordMap.put(i, chapterArray[i].replaceAll("\\W", "").trim());}
}
```

(2) It removes from the TreeMap words that match words in the repository of words that are not places. Because TreeMaps cannot read from and modified in the same runtime session, it is necessary here to use a *copy* of the TreeMap along with the original:

```
Map<Integer, String> placeWordMapCopy = new TreeMap<>();
placeWordMapCopy.putAll(placeWordMap);
for (String value : placeWordMapCopy.values()) {
        if (notPlacesList.contains(value.toLowerCase().trim())) {
        for (Entry<Integer, String> entry : placeWordMapCopy.entrySet()) {
                if (entry.getValue().equals(value)) {
```

```
            int key = entry.getKey();
            placeWordMap.remove(key, value); }}}}
```

(3) Once the TreeMap has been filtered, it is then written to the output file in the project folder:

```
for (Entry<Integer, String> entry : placeWordMap.entrySet()) {
        int key = entry.getKey();
        String value = entry.getValue();
        String line = key + "\t" + value;
        BufferedWriter bw = new BufferedWriter(new FileWriter(new
        File(chapterPlacesFileAddress), true));
        bw.write(line); bw.newLine(); bw.close();}
```

The *PlaceGetter* does not produce a final product, but a working file which will inevitably include lines which must be removed and others which must be edited. In order to assist the reader in determining the context and meaning of the extracted words, the SentenceGetter() method produces another file with the entire sentence in which the words saved in the previous method are located. However, the code ensures that any sentence that meets this criteria is appended to the file *only once*. This method declares variables to identify the boundaries of a sentence. It uses a **for** loop to read each line of the file produced by WordGetter(). It then moves through the array of chapter words in two directions to find the beginning and the end of the sentence in which the word is located with the following code:

```
for (line = br.readLine(); line != null; line = br.readLine()) {
        String sentencetoFind = "";
        for (int i1 = lineArrayInt; i1 < chapterArray.length; i1++) {
                if (chapterArray[i1].contains(fullStop)) {
                sentenceEnd = i1;
                break; }}

        for (int i1 = lineArrayInt; i1 >= 0; i1--) {
                if (chapterArray[i1].contains(fullStop)) {
                sentenceStart = i1 + 1; break;}}}
```

Then another for loop traverses from beginning to end to construct the sentence, assigning it to a variable:

```
for (int i1 = sentenceStart; i1 <= sentenceEnd; i1++) {
        sentencetoFind = sentencetoFind + " " + chapterArray[i1].trim();}
```

Finally the sentence is appended to a file and the method advances to the next line of the file produced by WordGetter(). Notice that this code advances the method to the end of the sentence. Once a sentence is appended to the output file, the method moves beyond it so that it will only be appended once:

```
line = br.readLine();
lineArray = line.split("\t");
lineArrayInt = new Integer(lineArray[0]);
if (lineArrayInt < sentenceEnd) {
        lineArrayInt = sentenceEnd;}
```

SoundGetter is structured identically to *PlaceGetter*, except that it filters words through a different repository of words that are sounds.

Next in the system is a trio of objects that operate on the output files produced by the preceding objects rather than the primary target files. These objects are called “Finishers” because they function to reformat the outputs according to certain requirements of Cage’s score, or to facilitate users in developing the performance.

MesosticFInisherAdjacentWordAdder creates a file in the project folder that adds words adjacent in the target text to the extracted word to mesostic lines up to a maximum number of characters either side of the word. The maximum number of characters is set according to a hardcoded parameter equal to 43, as Cage suggested in the score. However, this is not the only constraint on the number of words to be added either side of the mesostic word, because the preceding or following mesostic word may be located in the target text less than 43 characters from the current mesostic word. Code which stops the method from writing the same words on successive lines in the output file is included in this method. For a given line, the method writes the maximum number of words before the mesostic word, but stops writing words after it if the next mesostic word is reached before advancing 43 characters:

```
while ((length <= 43 && (index - indexPrevious) > 1) && index > lastWord+1) {
        wordsToAddBefore = chapterArray[index -
        1].toLowerCase().replaceAll("\\W", "").trim() + " " + wordsToAddBefore;
        index- -;
```

```
            length = wordsToAddBefore.length();
            }
        String wordsToAddAfter = "";
        length = 0;
        index = new Integer(lineContent[0]);
        while (length <= 43 && (indexNext - index) > 1) {
            wordsToAddAfter = wordsToAddAfter + " "
            + chapterArray[index + 1].toLowerCase().replaceAll("\\W", "").trim();
            index++;
            length = wordsToAddAfter.length();
        }
```

The method then constructs the line out of its parts and writes it to the output file:

```
        mesosticLine = lineContent[0] + "\t" + wordsToAddBefore + lineContent[1] +
        wordsToAddAfter;
        BufferedWriter bw = new BufferedWriter(new FileWriter(new
        File(allMesosticsWithAdjacentFileAddress), true));
        bw.write(mesosticLine); bw.newLine(); bw.close();
```

MesosticFinisherCentred. Cage prescribed that the lines of mesostic poems be centred on the uppercase letter in the mesostic word, and this object formats a mesostics file accordingly with its single method. It declares a final **int** variable called spacesToAddMax, set in the system at 35 spaces, and initialises another **int** variable called spaceAdjuster. Using a **while** loop, The method traverses the lines of a mesostics file character by character, finds the uppercase letter in each line, and assigns to the spaceAdjuster variable the character index value of that uppercase letter:

```
        for (int i = 0; i < lineContent[1].length(); i++) {
            if (Character.isUpperCase(lineContent[1].charAt(i))) {
                spaceAdjuster = i; }}
```

Next, to centre this line on the uppercase letter, it uses the value (spacesToAddMax - spaceAdjuster) to add the correct number of whitespaces to the beginning of the line:

```
        for (int i = 0; i < spacesToAddMax - spaceAdjuster; i++) {
            lineContent[1] = " " + lineContent[1]; }
```

Finally, it writes the newly spaced line to an output file.

MesosticFinisherCollated. This object combines three indexed output files into one master file which contains mesostic lines, sound words, and place words, each placed on appropriate lines in order of their index value. This object features a nested collection of if/else statements which moves line by line through the three target files. First it defines three lines, on from each file:

- line1 is the line from the AllMesostics file
- line2 is the line from the ChapterSounds file
- line3 is the line from the ChapterPlaces file

The smallest of the three is found and written to the output file and the line of the file with the smallest is advanced for the next iteration of the while statement. Try blocks surround the statements which advance a line. Catch blocks accompany these try blocks to catch a NullPointerException which occurs where there are no further lines to advance in a file. Catch blocks continue process with code comparing the two remaining lines, finding and writing the smaller of the two to the output file. Additional try/catch blocks are nested within these catch blocks to catch the NullPointerException which occurs where there are no further lines to advance in one of the two remaining files. Nested catch blocks write all the remaining lines of the last remaining file to the output file. The conditional logic coded here is straightforward, but the structure is somewhat complex because of the necessity to “catch” the situation where one of the files has reached its last line and is no longer involved in comparison. The reader is spared code snippets here. The code follows the following nexted conditional logic pattern:

OUTER IF line1<line2
    INNER IF (line1<line2)<line3
        -->write line 1
        -->try (advance line 1) catch (compare line2 and line3)
    INNER ELSE IF Line1<Line3<Line2
        -->write line 1
        -->try (advance line 1) catch (compare line2 and line3)
    INNER FINAL ELSE line3<(line1<line2)
        -->write line 3
        -->try (advance line 3) catch (compare line1 and line2)
OUTER ELSE IF line2<line3
        -->write line 2
        -->try (advance line 2) catch (compare line1 and line3)

OUTER FINAL ELSE line3 smallest

-->write line 3

-->try (advance line3) catch (compare line1 and line2)

MesosticGUI. This object is constructed using the WindowBuilder application within the Eclipse IDE using Absolute Layout and CardLayout formats. Within a JFrame object lies a navigable hierarchy of JPanel objects or "panels":

(1) A Welcome Panel with links to setup, run, and view cards
(2) A Setup Panel for choosing the project directory, mesostic row, and target chapter
(3) A Run Panel to call methods in the system's core functional objects to create files
(4) A Finish Panel to call methods to format the mesostic files
(5) A View Panel to view files produced by the Run and Finish Panel functions

The code is in general straightforward, much of it generated automatically by using the Eclipse WindowBuilder Design Page, where GUI items can be built with objects accessed via a point and click palette of Layouts, Containers, and Components (including, for example JFileChooser).

Use of the GUI is clearly explained in the *MesosticMachine User Manual*. This document is held in Appendix 3.

One of the objects is highlighted here because of the important work it does in the implementation of the system's objects. The Set Project Folder Button on the Setup Panel does a lot of work in this implementation. It opens up a "directories only" JFileChooser. Once the directory has been chosen with this JFileChooser, it assigns all the output files created by the system to the directory and it appends the directory to the SetupInfo panel text area. The code for this is crucial for the operation of all of the instantiated objects that reside elsewhere in the GUI. Here is the relevant snippet:

```
if (fileChooserDirectory.showOpenDialog(null)
        ==JFileChooser.APPROVE_OPTION){
        directory = fileChooserDirectory.getSelectedFile().getAbsolutePath();
        textArea.append("Project folder: " + directory + "\n\n");
        mesosticRowFileAddress = directory + "\\MesoticRow.txt";
        allMesosticsFileAddress = directory + "\\AllMesostics.txt";
        allMesosticsWithAdjacentFileAddress = directory +
        "\\AllMesosticsPlusAdjacent.txt";allMesosticsFinalCenteredFileAddress
        = directory + "\\AllMesosticsFinalCentered.txt";
```

```
allMesosticsCollatedFileAddress = directory +
"\\AllMesosticsCollated.txt";
soundFileAddress = directory +
"\\Sounds and Places\\OEDSounds Final Tabs.txt";
chapterSoundsFileAddress = directory +
 "\\Sounds and Places\\ChapterSounds.txt";
chapterSoundsSentencesFileAddress = directory +
"\\Sounds and Places\\ChapterSoundsSentences.txt";
notPlaceFileAddress = directory +
"\\Sounds and Places\\Not A Place.txt";
chapterPlacesFileAddress = directory +
"\\Sounds and Places\\ChapterPlaces.txt";
chapterPlacesSentencesFileAddress = directory +
 "\\Sounds and Places\\ChapterPlacesSentences.txt";
panelSetup.setVisible(true);
panelWelcome.setVisible(false);
} else if (fileChooserDirectory.showOpenDialog(null) ==
JFileChooser.CANCEL_OPTION) {
fileChooserDirectory.setVisible(false);
panelSetup.setVisible(false);}}
```

5.3 Testing Strategy

This system was developed through continuous testing of outputs printed to the console window in Eclipse. This type of testing, where code is written inside the “public static void main()” method in a Java “tester object“ compares *expected* output from statements, methods, and components to *actual* output produced in the console window. This is a “traditional” approach, but it has proven its effectiveness for this system. A test was devised for the statements in every component of conditional logic to verify that expected and actual outcomes were aligned. Some of the classes developed within the framework of a static method are held in separate Java packages submitted alongside the main system package accompanying this dissertation. A limited amount of retrospective unit testing using the JUnit library was also conducted. Code from some of these tester classes can be viewed in Appendix 5.

It should be clear to the reader at this point that the author has fully capitalised on the experience of conducting all of the functions of this system “by eye and hand”, an experience which generated an

extremely useful body of test data from Ciaran Carson's *The Star Factory*: a full set of mesostics, a full set of sounds, and a full set of places. Once the text of *The Star Factory* was reformatted as a series of text files and cleaned, the author was in possession of both test data and *expected* system outcome data (the actual mesostics created by hand, the lists of sounds and places, and their locations in the text). This material was used continuously to test the software system as it was developed. However, because these data were created by eye and hand, they are likely to have errors embedded in them. The hand-made mesostics cannot be treated as pure "actual" test data for the MesosticMachine. Each method tests the other. Not surprisingly, by running tests on the first pages of *The Star Factory*, the *MesosticMachine* can be shown to be functioning more accurately than the hand and eye process conducted in 2012 (which produced three errors in the very first mesostic!). The *MesosticMachine* has yet to be tested on the entire text of *The Star Factory*, but it can be safely be predicted that there will be many differences in the set of poems that it produces compared to the *Owenvarragh* text.

Chapter 6: Evaluation and Conclusion

By automating algorithms in John Cage's score, this project significantly enhances the ability of an artist to bring the intended musical performances of literary works to fruition. The system produces sets of mesostics, which become the central spine of the performance, in a range of formats. It also produces word lists to facilitate the artist's search for sounds and places of the literary work. The development strategy adopted was successful, with the system completed within a week of the deadline for submission. The author developed valuable skills handling Java functionality relating to the manipulation of textual data, including functionality for String objects, Array objects, the Java Collections Framework, Webdrivers, and reading to and writing from files. And in order to make proper use of these tools, the author climbed a steep (though at times frustratingly difficult) learning curve to become more adept with the strategies of code creation and testing. A valuable outcome of this project is the authors much greater understanding of the behaviour of for and while loops, of statements operationalising conditional logic, and of the importance of the structure and scope of variables used in the code. The project also presented a valuable opportunity to move beyond Java and development an understanding of syntax in Python, as part of the effort to identify sound words in the WordNet database.

The experience was also disappointing in some ways. As the development journal shows, implementing each of the significant components of the design in Java code was perhaps more time-consuming than would be expected of a professional programmer, given the modest scale of the requirements. Problems remain in the code, particularly with regard to the behaviour of loops as they reach their terminus and or/the terminus of the object they are traversing. The project achieved what it set out to do, but pressure of time prohibited the presentation of a more fully developed and tested system. As explained in Chapter 3, the system was developed to facilitate the particular interpretation of the score adopted for the *Owenvarragh* project, but a number of use case scenarios suggested by Cage in the score – for example different ways to deal with reaching the end of a file, or the end of mesostic row, or for dealing with unmanageable long lists of places or sounds – were not included in the requirements specification and not catered for by the system. The interface between the text outputs and word processing programs other than Notepad must be developed. Functionality for exact placement of sound and place markers in an mp3 sound file of a mesostics recitation, a user requirement raised by Úna Monaghan in my elicitation interview with her, was also left for future development, though the "collated" mesotics output file takes a useful step in this direction. .

Pressure of time also prohibited the production of a complete set of mesostics for *The Star Factory*, running the system to completion on every chapter of the book. To facilitate a complete set of mesostics, an additional "finishing" object would have been useful, one perhaps constructed using the "builder" design pattern to the mesostics files from each chapter and integrate them (updating the index values of each during the build) into a master file. Working on this implementation might have

uncovered other “scaling” problems to solve for users, but there was no time left to take these final steps.

With regard to the sounds and places functionality, there is still work to do. The system will produce outputs that are only as good as the comparator repositories used. At the time of submission the system is functioning adequately, but the repositories themselves, particularly the sound repository file, needs further attention. The experience of testing the first chapters of *The Star Factory* has demonstrated that the file has far too many words and produces too many false positives. Very many of the adjectives appear to be superfluous, both because they are multivalent in meaning and rarely used in the context of a sound sense, and because when they are used in the context of a sound sense, the word they modify is also a sound. Similar issues remain with regard to excessive false positive results in the places output files. These will take some time and much more testing to refine. For now, however, a significant advance over scrutinising a text by eye and hand has been provided.

At the start of the project it was envisioned that the system could be unleashed on a larger sample of literary works. Doing so might uncover the usefulness of the *MesosticMachine* for literary research and artistic production beyond the context of implementing the Cage score. As mentioned above, the system allows for a quick comparison of the scale of longer versus shorter works. It also allows quick comparison of the use of different mesostic rows for a given work (author’s name versus title). Running the system on a collection of works brings a research project to compare the poetic quality of different works, and the relative noisiness of a potential performance of different works, into view. The system might also provide further understanding of the “bunching effect” of mesostic creation, and sound and place location, caused by the uneven distribution of extracted words in a text. With regard to *The Star Factory*, Dowling and Monaghan reported:

> It gradually became apparent that first the Y, and then the F, would give the desired result. The process produced mesostics drawn very unevenly from the text, with the lines THESTAR using words quite close to each other in the text, followed by increasingly lengthy gaps between R and F. Another “bunching” effect occurred with the letters ACTOR, followed by another increasingly lengthy gap before Y, the last letter of the mesostic. Keeping an index of syllables using S proved quite early on to be pointless, because of all the unique syllables in pluralized words. In contrast, the letter F proved to be relatively rare, and the words using an F which also have an A following that F were encountered (and skipped over) in a high percentage of instances.[26]

A useful tool has now exists to realise John Cage’s score. Further refinement of this tool, developing solutions for scaling up its functions, and investigating its usefulness for literary research, are the next steps.

[26] Dowling and Monaghan, “Creating a Circus,” p. 10.

Bibliography

Bugayenko, Yegor. A Blog About Computers, http://www.yegor256.com/.

Cage, John. Cage: *Roaratorio; Laughtears—Cage, Shöning, Heaney, Ennis, et. al; Writing for the Second Time Through Finnegan's Wake* (Mode Records, 28/29, New York, 1992).

Carson, Ciaran. *The Star Factory* (Granta, London, 1997).

Deitel, Paul and Harvey. *Java: How to Program; Early Objects* (London: Perason, 2015).

Dowling, Martin. "Rhythm, Rhymes, and Pleasure: Mesostics on Ciaran Carson's *The Star Factory*," *Journal of Black Mountain College Studies*, Special John Cage Issue, vol. 4, 2013(www.blackmountainstudiesjournal.org)

Dowling, Martin. YouTube Channel www.youtube.com/playlist?list=PLOa0O4aBIs3UPWfV1l-A0WKO25866Duzz

Dowling, Martin, and Úna Monaghan, "Creating a Circus of Words, Music, and Sound: From *Roaratorio* to *Owenvarragh*," *Journal of Black Mountain College Studies*, Special John Cage Issue, vol. 4, 2013 (www.blackmountainstudiesjournal.org)

Monaghan, Úna McAlister. *New technologies and experimental practices in contemporary Irish traditional music: a performer-composer's perspective* (PhD, Queen's University of Belfast, 2015).

Natural Language Toolkit Website, http://www.nltk.org/.

Oracle Website, https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html.

Oracle Website, https://docs.oracle.com/javase/6/docs/api/java/lang/String.html.

Oxford English Dictionary Website, http://www.oed.com/.

Stanford Natural Language Processing Group Website, https://nlp.stanford.edu/.

Wordnet Website: http://wordnet.princeton.edu/.

Appendix 1. The John Cage Score

"_____, _____ _____
(title of composition) (article) (adjective)
Circus On _____"
(title of book)

Means for translating a book into a performance without actors, a performance which is both literary and musical or one or the other, by John Cage.

Edinburgh, September 1979

For Klaus Schöning

1. Choose a book. If it is not in the public domain, obtain permission for its use from those owning the copyright. Failing that, make step 2 in such a way that no relationships of words occur that occur in the original, or encode your text so that the original words are not represented by themselves (change the title of your work accordingly).

2. Taking the name of the author and/or the title of the book as their subject (the row), write a series of mesostics beginning on the first page and continuing to the last. Mesostics means a row down the middle. In this circumstance a mesostic is written by finding the first word in the book that contains the first letter of the row that is not followed in the same word by the second letter of the row. The second letter belongs on the second line and is to be found in the next word that contains it that is not followed in the same word by the third letter of the row. Etc. If a shorter rather than longer text is desired, keep an index of the syllables used to represent a given letter. Do not permit for a single appearance of a given letter the repetition of a particular syllable. Distinguish between subsequent appearances of the same letter. Other adjacent words from the original text (before and/or after the middle word, the word including a letter of the row) may be used according to taste, limited, say to forty-three characters to the left and forty-three to the right, providing the appearance of the letters of the row occurs in the way described above. Omit punctuation and capitalize the row, reducing all other capitals to lower case. If at the end of the book or a chapter of it a mesostic is not complete, leave it incomplete or complete it by returning to the first page of the book or chapter and continuing your search for words containing the necessary letters. Having completed the series of mesostics, identify each line by page and line of the original from which it came. Make a tape recording of the recital of the text using speech, song, chant, or Sprechstimme, or a mixture or combination of these. Ascertain its time-length. Subtract that from a total program length, and distribute the thus-determined silence between large parts and chapters of parts and at the beginning and end of the tape. You then

have a ruler in the form of a typed or printed text and in the form of a recited text, both of them measurable in terms of space (page and line) and time (minute and second), by means of which the proper position (see 5 below) of sounds (see 3 and 4 below) may be determined.

3. Make a list of places mentioned in the book and a list for each of the pages and lines where the mention is made for each. If the list once made is unmanageably long, reduce in some chance-determined way, e.g. to a number equal to the number of pages in the book.

4. Make a list of sounds mentioned in the book and a list for each of the pages and lines where the mention is made. If the list once made is unmanageably long and baffling because of the large number of kinds of sounds, establish families of sounds and extract from the whole list those related to certain of these.

5. Collect as many recordings as possible made in the places mentioned (3) and of sounds mentioned in the book (4). Re-record them in stereo one at a time on a multitrack tape at proper points in time (see 2 above) following chance determinations (a) of stereo position (using a large number of positions as can be conveniently distinguished), b) of relative duration (short, medium, long), c) of attack (roll on, fade in, switch on), d) of successions of loudness or loudness, and e) of decay (roll off, fade out, switch off). Placing transparent graph paper over the typed or printed text, inscribe the placement of each tape horizontally with respect to words in the text, and vertically with respect to available tracks. To do the work described in this paragraph, two people, one of them a sound engineer, must work together. When a multitrack tape is filled up, proceed to another. Given the circa half-hour length of multitrack tapes, divide the available studio time (reserving several days at the end for final mixing and assembling the tapes) first by the number of half-hours and then by two (place sounds and text sounds). That will permit the planning of a work schedule so that an equal amount of time may be devoted to each aspect and each part of the composition. The work is finished when the available studio time runs out.

6. Using recordings of relevant musics (in performances by soloists) or composed variations of such solos, make a chance determined total program for each having at least twice as much silence as music. Superimpose these on a multitrack tape to make a circus of relevant musics.

7. Without erasing any reduce the collection of multitrack tapes to a single one. The material is then in a plurality of forms. At one extreme, each pair of tracks can be given its own stereo sound system. At the other (suitable for radio transmission) all tracks can be heard through a single stereo system. Other uses of the material are also available so that the various “layers” (2, 3, 4, 6) can be heard alone or in any combinations, and some of them (2, 6) as recorded, or, when some or all of the performers are available, live.

Appendix 2. Score as Requirements Specification

1. Choosing a book.

    1.1. The user shall choose a book that is in the public domain, and transfer the book to a file in UTF 8 format with a .txt extension.
    1.2. If the book is not in the public domain, the user shall obtain permission for its use from those owning the copyright.
    1.3. Failing that, the user shall make section 2 of this document in such a way that no relationships of words occur that occur in the original, or encode your text so that the original words are not represented by themselves (change the title of your work accordingly).

2. Making Mesostics.

    2.1. The system shall allow the user to input and save the name of the author and/or the title of the book as the subject of the row.
    2.2. The system shall allow the user to create and save to a .txt file in UTF8 format a series of mesostics beginning on the first page of the book and continuing to the last.
    2.3. If a shorter rather than longer text is desired by the user, the system shall allow the user to keep an index of the syllables used to represent a given letter, not permitting for a single appearance of a given letter the repetition of a particular syllable, and distinguishing between subsequent appearances of the same letter.
    2.4. The system shall allow the user to add adjacent words from the original text (before and/or after the middle word, the word including a letter of the row) according to taste, limited, say to forty-three characters to the left and forty-three to the right, providing the appearance of the letters of the row occurs in the way described above.
    2.5. The system shall remove punctuation and capitalize the row, reducing all other capitals to lower case.
    2.6. If at the end of the book or a chapter of it a mesostic is not complete, the system shall leave it incomplete (or complete it by returning to the first page of the book or chapter and continuing your search for words containing the necessary letters).
    2.7. The system shall identify each mesostic line by page and line of the original word from which it came.

3. Making a list of places

    3.1. The system shall make a list of places mentioned in the book.
    3.2. The system shall make a list for each of the pages and lines where the mention is made for each place.

3.3. If the list once made is unmanageably long, the system shall provide functionality to reduce the list in some chance-determined way, e.g. to a number equal to the number of pages in the book.

4. Making a list of sounds.

4.1. The system shall allow the user to make a list of sounds mentioned in the book
4.2. The system shall make a list for each of the pages and lines where the mention of each sound is made.

Appendix 3. MesosticMachine User Manual

This document is a guide to using the MesosticMachine graphical user interface (GUI) to set up a project and create mesostics, lists of sounds and places, and a number of differently structured files containing this extracted information. The system requires the user to proceed through the steps below in the following order:

- steps 1.1-1.4 *must* be completed before running the GUI
- steps 2.1-2.5 *must* be completed every time the GUI is run, and before any further steps are completed. If this is not done, the GUI will display an error message, and the user must close the GUI and begin again.
- steps 3.1-3.3 *must* be completed before step 4.X
- steps 3.1-3.6 *must* be completed before step 4.X

5. Preparing and maintaining the project folder environment.

    5.1. Create a project folder hierarchy which has a main project folder, a sounds and places subfolder, and a syllable repository subfolder on a disk accessible to the MesosticMachine system. The following setup is recommended for a Windows OS:
        5.1.1. The default directory for the Mesostic Machine: “C:\Users”
        5.1.2. The main project folder “C:\Users\ThisUser\Documents\Mesostic Machine”. The User will copy the current target chapter (kept in the Chapter Repository) in this folder, and after the system runs, delete this chapter and move the output files to the Mesostics Repository SubFolder.
        5.1.3. The subfolders in the main project folder: “~\Chapter Repository” which contains all of the target chapters, “~\Sounds and Places” which will contain output files, “~\Syllable Repository” which will contain output files, and “~\Mesostics Repository” which will contain output files, and “Reference Documents”.
    5.2. Format the target book, or each chapter of the book if appropriate, as a UTF8 text file, and store in a book repository folder.
    5.3. Place the book, or the first chapter file, in the project folder
    5.4. If working with separate chapters, when all steps are complete with the first chapter, remove the chapter and all outputs to another folder, and place the next chapter file in the project folder.

6. Setting up the system

    6.1. Open the Eclipse IDE, the MesosticMachine Project in its workspace, and the mesosticSystem Package within the Project, and the MesosticGUI Class within the Package.
    6.2. Run the MesosticGUI. The Welcome Panel is displayed.
    6.3. Click the Setup Button. The Setup Panel is displayed.

6.4. Click the Setup Project Folder Button. A Windows Explorer emulator is displayed on the page C:\users. Browse to the appropriate folder (see 1.1 above) and click the Open button. The directory variable is set and the Setup Panel is displayed.
6.5. Click on the Set Chapter File Button. A Windows Explorer emulator displays the project folder. Click on the appropriate chapter file (see 1.2 -1.4 above) and click the Open button. The chapterFileAddress variable is set and the Setup Panel is displayed.
6.6. Click on the Set Mesostic Row button. A new panel with a text field is displayed. Type the desired mesostic row in the field and click the Submit button. The mesostic row is appropriately formatted for the system, the MesosticRow.txt file is created in the project folder, and the mesostic row is written to that file. The Setup Panel is Displayed.
6.7. Click on the Your Setup Info button. The results of steps 2.4-2.6 are displayed. Check if the information is correct. Press OK. The Setup Panel is displayed. If there are errors, repeat the steps 2.4-2.6 as necessary. If the information is correct, click on the Done Button. The Welcome Panel is displayed.

7. Making Mesostics.

7.1. From the Welcome Panel, click on the Run Button. The Run Panel is displayed. Click on the Make Mesostics Button. The Make Mesostics Panel is displayed.
7.2. Decide on a shorter or longer version of the mesostics process.
7.3. If a shorter version is required, click on the Shorter Version Button. The AllMesostics.txt file will be created in the project folder and filled with mesostics built with the syllable filter.
7.4. If a longer version is required, click on the Longer Version Button. The AllMesostics.txt file will be created in the project folder and filled with mesostics built without the syllable filter.
7.5. Click on the Done Button. The Run Panel is Displayed.

8. Making a list of places

8.1. While the Run Panel is displayed, click on the Get Places Button. The ChapterPlaces.txt and ChapterPlacesSentences.txt files are created in the Sounds and Place subfolder (see step 1.1). Extracted words and their index values are appended to the ChapterPlaces.txt file. The sentence in which those words are located along with the index value of the word are appended to the ChapterPlaces.txt file.
8.2. Click the Done Button. The Run Panel is displayed.
8.3. Navigate to the Welcome Panel and proceed to step 7. Open and edit the places files to prepare them for collation (step 6.5).

9. Making a list of sounds.

9.1. While the Run Panel is displayed, click on the Get Sounds Button. The ChapterSounds.txt and ChapterSoundsSentences.txt files are created in the Sounds and Place subfolder (see step 1.1). Extracted words and their index values are appended to the ChapterSounds.txt file. The sentence in which those words are located along with the index value of the word are appended to the ChapterSounds.txt file.

9.2. Click the Done Button. The Run Panel is displayed

9.3. Navigate to the Welcome Panel and proceed to step 7. Open and edit the soounds files to prepare them for collation (step 6.5).

10. Finishing the Mesostics Files

10.1. From the Welcome Panel, click on the Finish Button. The Finish Panel is displayed.

10.2. Click on the Add Adjacent Word Button. The AllMesosticsPlusAdjacent.txt file is created in the project folder and mesostic lines with a maximum number of adjacent words are appended to this file. The user will see that the lines in this file have a maximum number of adjacent words added. It is suggested that the user tailor this file by reducing the number of adjacent words either side of the mesostic row “according to taste”, as Cage instructs in the score, before proceeding. This is easily done by navigating to the View Panel and opening the file with Notepad.exe.

10.3. Navigate to the Welcome Panel and proceed to step 7. Open the AllMesosticsPlusAdjacent.txt file, delete adjacent words “according to taste”, and save the file. Navigate to the Welcome Panel. Click on the Finish Button.

10.4. Click on the Center Mesostic Lines Button. The AllMesosticsFinalCentred.txt file is created in the project folder and mesostic lines centred on the uppercase letter in the line are appended to this file. Click on the Done Button. The Welcome Panel is Displayed. Click on the Finish Button.

10.5. Click on the Collate Sounds and Places Button. The AllMesosticsCollated.txt file is created in the project folder and indexed lines of sound and place words are collated with the mesostic lines according to index value are appended to this file. Click on the Done Button. The Welcome Panel is Displayed.

11. Viewing and editing the system output files

11.1. From the Welcome Panel, click on the View Button. The View Panel is displayed.

11.2. Click on the Open Button. A Windows Explorer emulator displays the project folder and its .txt files. Click on any file to open it in with Notepad.exe. View and/or edit the file.

If the user is working with multiple chapters of a target text, once these steps have been taken with the first chapter, the chapter file and its output files should be removed from the project folder and saved in a repository for further processing. Then the user may resume with step 1.1 with next chapter in the book.

Appendix 4. Python NLTK Routines for Querying Synsets and Hyponyms in WordNet

Python syntax

// = comment line

= comment line

>>> = Python statement

[outputs in brackets]

WordNet Python Routine

```
>>>import nltk

>>>nltk.download()

>>>from nltk.corpus import wordnet as wn

#startup nltk and load wordnet

>>>wn.synsets('sound')

#return a list of synsets for 'sound'

>>>for synset in wn.synsets('sound'): print(synset.name(), synset.lemma_names(), synset.definition(), sep = '\t')

#return the name, the lemmas, and the definition of each synset for 'sound'

>>>SSyes = list([wn.synset('sound.n.01'), wn.synset('sound.n.02'), wn.synset('sound.n.04'), wn.synset('sound.v.02'), wn.synset('sound.v.03'), wn.synset('sound.v.04'), wn.synset('sound.v.06'), wn.synset('voice.v.02'), wn.synset('audio.n.01'), wn.synset('phone.n.02'),])

#create a list of relevant synsets

>>>for synset in SSyes: print(synset.name(), synset.lemma_names(), synset.definition(), sep = '\t')

#show the set of relevant synsets

>>>SSno = list([wn.synset('sound.n.03'), wn.synset('strait.n.01'), wn.synset('sound.n.08'), wn.synset('sound.v.01'), wn.synset('fathom.v.02'), wn.synset('sound.a.01'), wn.synset('healthy.s.04'), wn.synset('sound.a.03'), wn.synset('good.s.17'), wn.synset('reasoned.s.01'), wn.synset('legal.s.03'), wn.synset('sound.s.07'), wn.synset('heavy.s.26'), wn.synset('sound.s.09')])

#create a set of irrelevant synsets

>>>length = len(SSyes)

#return a count of relevant synsets

#return the next level hypernyms for each of these synsets.

>>>for synset in SSyes: print(synset.name(), synset.hypernyms, sep = '\t'

>>>for x in range(0, length - 1):

        set([i for i in SSyes[x].closure(lambda s:s.hyponyms())])

#return all the hypernyms for each of these synsets
```

#work with lemmas instead of synsets??

>>>SN1lemma = set(sorted(lemma.name() for synset in wn.synset('sound.n.01').hyponyms() for lemma in synset.lemmas()))

>>>words = [w for w in SN1lemma]

>>>wn.synsets(words[0])

returns: [Synset('unison.n.01'), Synset('unison.n.02'), Synset('unison.n.03')]

#a better search

>>>for synset in wn.synsets(words[0]): print(synset.name(), synset.lemma_names(), synset.definition(), sep = '\t')

>>>sorted(lemma.name() for synset in wn.synset('unison.n.01').hyponyms() for lemma in synset.lemmas())

returns: [], likewise for the other two

>>>for synset in wn.synsets('dog'):

    for lemma in synset.lemmas():

        print lemma.name()

>>>wn.synset('dog.n.1').lemma_names()

Routine for Querying Synsets of "Sound"

//Startup routine

>>>import nltk

>>>nltk.download()

>>>from nltk.corpus import wordnet as wn

//return a list of synsets for 'sound'

>>> wn.synsets('sound')

//better yet, print out the name, the lemmas, and the definition of each synset for 'sound'

>>> for synset in wn.synsets('sound'): print(synset.name(), synset.lemma_names(), synset.definition(), sep = '\t')

Sorting the output by relevance

//MEANINGS RELATING TO AUDITORY PHENOMENA

sound.n.01	['sound']	the particular auditory effect produced by a given cause
sound.n.02	['sound', 'auditory_sensation']	the subjective sensation of hearing something
sound.n.04	['sound']	the sudden occurrence of an audible event
sound.v.02	['sound', 'go']	make a certain noise or sound
sound.v.03	['sound']	give off a certain sound or sounds

sound.v.04 ['sound'] announce by means of a sound

voice.v.02 ['voice', 'sound', 'vocalize', 'vocalise'] utter with vibrating vocal chords

sound.v.06 ['sound'] cause to sound

audio.n.01 ['audio', 'sound'] the audible part of a transmitted signal

phone.n.02 ['phone', 'speech_sound', 'sound'] (phonetics) an individual sound unit of speech without concern as to whether or not it is a phoneme of some language

//OTHER MEANINGS

sound.n.03 ['sound'] mechanical vibrations transmitted by an elastic medium (i.e. "ulstrasound")

strait.n.01 ['strait', 'sound'] a narrow channel of the sea joining two larger bodies of water

sound.n.08 ['sound'] a large ocean inlet or deep bay

sound.v.01 ['sound'] appear in a certain way

fathom.v.02 ['fathom', 'sound'] measure the depth of (a body of water) with a sounding line

sound.a.01 ['sound'] financially secure and safe

healthy.s.04 ['healthy', 'intelligent', 'levelheaded', 'level-headed', 'sound'] exercising or showing good judgment

sound.a.03 ['sound'] in good condition; free from defect or damage or decay

good.s.17 ['good', 'sound'] in excellent physical condition

reasoned.s.01 ['reasoned', 'sound', 'well-grounded'] logically valid

legal.s.03 ['legal', 'sound', 'effectual'] having legal efficacy or force

sound.s.07 ['sound'] free from moral defect

heavy.s.26 ['heavy', 'profound', 'sound', 'wakeless'] (of sleep) deep and complete

sound.s.09 ['sound'] thorough

Routine to return hyponyms for sound.n.01

```
>>> soundN1 = wn.synset('sound.n.01')
>>> soundsN1 = soundN1.hyponyms()
>>> sorted(lemma.name() for synset in soundsN1 for lemma in synset.lemmas())
```

//or just do it on one line

```
>>> sorted(lemma.name() for synset in wn.synset('sound.n.01').hyponyms() for lemma in synset.lemmas())
```

Output: ['noisiness', 'racketiness', 'ring', 'unison', 'voice']

Routine to return hyponyms for sound.n.02

```
>>> soundN2 = wn.synset('sound.n.02')
>>> soundsN2 = soundN2.hyponyms()
```

```
>>> sorted(lemma.name() for synset in soundsN2 for lemma in synset.lemmas())
['dissonance', 'dub', 'euphony', 'music', 'music', 'noise', 'pure_tone', 'racket', 'tone']
```

Routine to return hyponyms for sound.n.03

```
>>> soundN3 = wn.synset('sound.n.03')
>>> soundsN3 = soundN3.hyponyms()
>>> sorted(lemma.name() for synset in soundsN3 for lemma in synset.lemmas())
['ultrasound']
```

Routine to return hyponyms for sound.n.04

```
>>> soundN4 = wn.synset('sound.n.04')
>>> soundsN4 = soundN4.hyponyms()
>>> sorted(lemma.name() for synset in soundsN4 for lemma in synset.lemmas())
['beat', 'beep', 'bell', 'birr', 'bleep', 'bombilation', 'bombination', 'bong', 'buzz', 'chink', 'chirp', 'chirrup',
'chorus', 'click', 'click-clack', 'clink', 'clip-clop', 'clippety-clop', 'clop', 'clopping', 'clump', 'clumping',
'clunk', 'clunking', 'cry', 'ding', 'drip', 'dripping', 'drum', 'drum_roll', 'drumbeat', 'footfall', 'footstep',
'gargle', 'gurgle', 'jangle', 'jingle', 'knock', 'knocking', 'murmur', 'murmuration', 'murmuring', 'mussitation',
'mutter', 'muttering', 'noise', 'paradiddle', 'pat', 'patter', 'peal', 'pealing', 'ping', 'plunk', 'pop', 'popping',
'purr', 'quack', 'quaver', 'rap', 'rataplan', 'ring', 'ringing', 'roll', 'roll', 'rolling', 'rub-a-dub', 'sigh', 'skirl',
'song', 'step', 'strum', 'susurration', 'susurrus', 'swish', 'tap', 'tapping', 'throbbing', 'thrum', 'thud', 'thump',
'thumping', 'thunk', 'tick', 'ticking', 'ting', 'tinkle', 'tintinnabulation', 'toll', 'toot', 'tootle', 'trample',
'trampling', 'twang', 'twitter', 'vibrato', 'voice', 'vroom', 'whack', 'whir', 'whirr', 'whirring', 'whistle',
'whistling', 'whiz', 'zing', 'zizz']
```

Routine return hyponyms for audio.n.01

```
>>> audioN1 = wn.synset('audio.n.01')
>>> audiosN1 = audioN1.hyponyms()
>>> sorted(lemma.name() for synset in audiosN1 for lemma in synset.lemmas())
```

No output!

Routine to return hyponyms for phone.n.02

```
>>> phoneN2 = wn.synset('phone.n.02')
>>> phonesN2 = phoneN2.hyponyms()
>>> sorted(lemma.name() for synset in phonesN2 for lemma in synset.lemmas())
['consonant', 'glide', 'orinasal', 'orinasal_phone', 'phoneme', 'semivowel', 'sonant', 'voiced_sound',
'vowel', 'vowel_sound']
//return hyponyms for sound.v.02
>>> soundV2 = wn.synset('sound.v.02')
```

```
>>> soundsV2 = soundV2.hyponyms()

>>> sorted(lemma.name() for synset in soundsV2 for lemma in synset.lemmas())

['babble', 'bang', 'beat', 'beat', 'beep', 'birr', 'blare', 'bleep', 'blow', 'blow', 'bombilate', 'bombinate',
'boom', 'boom', 'boom_out', 'bubble', 'burble', 'buzz', 'chatter', 'chime', 'chink', 'chug', 'clang', 'clangor',
'clangor', 'clangour', 'clank', 'claxon', 'click', 'click', 'clink', 'clop', 'clump', 'clunk', 'crack', 'crack', 'crash',
'din', 'drone', 'drum', 'echo', 'glug', 'grumble', 'guggle', 'guggle', 'gurgle', 'gurgle', 'honk', 'hum', 'knock',
'knock', 'lap', 'make_noise', 'noise', 'patter', 'peal', 'ping', 'ping', 'pink', 'pink', 'pitter-patter', 'plunk', 'pop',
'purr', 'rap', 'rattle', 'resonate', 'resound', 'resound', 'reverberate', 'ring', 'ring', 'ripple', 'roll', 'rumble',
'rustle', 'sing', 'skirl', 'slosh', 'slush', 'snap', 'snarl', 'splash', 'splat', 'splosh', 'squelch', 'swish', 'swoosh',
'swosh', 'tap', 'thrum', 'thrum', 'thud', 'thump', 'tick', 'tick', 'ticktack', 'ticktock', 'ting', 'tink', 'tinkle', 'toot',
'trump', 'twang', 'tweet', 'twirp', 'vibrate', 'whir', 'whirr', 'whish', 'whistle', 'whistle', 'whiz', 'whizz']
```

Routine to return hyponyms for sound.v.03

```
>>> soundV3 = wn.synset('sound.v.03')

>>> soundsV3 = soundV3.hyponyms()

>>> sorted(lemma.name() for synset in soundsV3 for lemma in synset.lemmas())

['cackel', 'dissonate', 'pierce', 'play', 'speak']
```

Routine to return hyponyms for sound.v.04

```
>>> soundV4 = wn.synset('sound.v.04')

>>> soundsV4 = soundV4.hyponyms()

>>> sorted(lemma.name() for synset in soundsV4 for lemma in synset.lemmas())
```

No Output!

Routine to return hyponyms for voice.v.02

```
>>> voiceV2 = wn.synset('voice.v.02')

>>> voicesV2 = voiceV2.hyponyms()

>>> sorted(lemma.name() for synset in voicesV2 for lemma in synset.lemmas())

['chirk', 'quaver', 'waver']
```

Routine to return hyponyms for sound.v.06

```
>>> soundV6 = wn.synset('sound.v.06')

>>> soundsV6 = soundV6.hyponyms()

>>> sorted(lemma.name() for synset in soundsV6 for lemma in synset.lemmas())

['blow', 'clink', 'gong', 'knell', 'play', 'pop', 'prepare', 'ring', 'strum', 'thrum', 'ting', 'twang']
```

Routine for Querying Synsets and Hyponyms of "Noise"

```
>>> wn.synsets('noise')

[Synset('noise.n.01'), Synset('noise.n.02'), Synset('noise.n.03'), Synset('noise.n.04'),
Synset('noise.n.05'), Synset('randomness.n.02'), Synset('make_noise.v.01')]

>>> for synset in wn.synsets('noise'): print(synset.name(), synset.lemma_names(), synset.definition(),
sep = '\t')
```

NOISE SYSNSETS

noise.n.01 ['noise'] sound of any kind (especially unintelligible or dissonant sound)

noise.n.02 ['noise', 'dissonance', 'racket'] the auditory experience of sound that lacks musical quality; sound that is a disagreeable auditory experience

noise.n.03 ['noise', 'interference', 'disturbance'] electrical or acoustic activity that can disturb communication

noise.n.04 ['noise'] a loud outcry of protest or complaint

noise.n.05 ['noise'] incomprehensibility resulting from irrelevant information or meaningless facts or remarks

randomness.n.02 ['randomness', 'haphazardness', 'stochasticity', 'noise'] the quality of lacking any predictable order or plan

make_noise.v.01 ['make_noise', 'resound', 'noise'] emit a noise

Routine to eturn hyponyms for noise.n.01

```
>>> noiseN1 = wn.synset('noise.n.01')

>>> noisesN1 = noiseN1.hyponyms()

>>> sorted(lemma.name() for synset in noisesN1 for lemma in synset.lemmas())
```

['bam', 'bang', 'banging', 'bark', 'blare', 'blaring', 'blast', 'boom', 'brouhaha', 'cacophony', 'chatter', 'chatter', 'chattering', 'chattering', 'chug', 'clack', 'clamor', 'clang', 'clangor', 'clangoring', 'clangour', 'clank', 'clap', 'clap', 'clash', 'clatter', 'crack', 'cracking', 'crackle', 'crackling', 'crash', 'creak', 'creaking', 'crepitation', 'crunch', 'din', 'ding-dong', 'eruption', 'explosion', 'fizzle', 'grate', 'grinding', 'grumble', 'grumbling', 'grunt', 'hiss', 'hissing', 'howl', 'hubbub', 'hum', 'humming', 'hushing', 'katzenjammer', 'oink', 'pant', 'plash', 'plonk', 'plop', 'plump', 'racket', 'rale', 'rattle', 'rattling', 'report', 'rhonchus', 'roar', 'roaring', 'rumble', 'rumbling', 'rustle', 'rustling', 'scrape', 'scraping', 'scratch', 'scratching', 'scream', 'screaming', 'screech', 'screeching', 'scrunch', 'shriek', 'shrieking', 'shrilling', 'sibilation', 'sizzle', 'slam', 'snap', 'snap', 'snore', 'spatter', 'spattering', 'splash', 'splatter', 'splattering', 'splutter', 'sputter', 'sputtering', 'squawk', 'squeak', 'squish', 'stridulation', 'swoosh', 'thunder', 'thunder', 'uproar', 'whisper', 'whispering', 'whoosh']

Routine to return hyponyms for noise.n.02

```
>>> noiseN2 = wn.synset('noise.n.02')

>>> noisesN2 = noiseN2.hyponyms()

>>> sorted(lemma.name() for synset in noisesN2 for lemma in synset.lemmas())
```

No Output!

Routine to return hyponyms for noise.n.04

```
>>> noiseN4 = wn.synset('noise.n.04')
>>> noisesN4 = noiseN4.hyponyms()
>>> sorted(lemma.name() for synset in noisesN4 for lemma in synset.lemmas())
```

Routine to return hyponyms for make_noise.v.01

```
>>> makenoiseV1 = wn.synset('make_noise.v.01')
>>> makenoisesV1 = makenoiseV1.hyponyms()
>>> sorted(lemma.name() for synset in makenoisesV1 for lemma in synset.lemmas())
```

['backfire', 'blare', 'blast', 'brattle', 'clack', 'clatter', 'claxon', 'clitter', 'crackle', 'creak', 'crunch', 'drown_out', 'honk', 'howl', 'hum', 'jangle', 'jingle', 'jingle-jangle', 'purl', 'racket', 'ring_out', 'roar', 'scranch', 'scraunch', 'screak', 'scream', 'screech', 'sizzle', 'skreak', 'sough', 'squeak', 'stridulate', 'whine']

Appendix 5. Sample Test Classes

```
public class MesosticGUIObjectTester {


public static void main(String[] args) throws IOException, InterruptedException {


String directory = "C:\\Users\\Martin\\Documents\\MesosticsMachine\\Reference and Testing
Documents\\Test Files";

String mesosticRowFileAddress = directory + "\\MesoticRow.txt";;

String chapterFileAddress = directory + "\\01 RAGLAN STREET.txt";

String allMesosticsFileAddress = directory + "\\AllMesostics.txt";

String allMesosticsWithAdjacentFileAddress = directory + "\\AllMesosticsWithAdjacent.txt";

String allMesosticsFinalCenteredAddress = directory + "\\AllMesosticsFinalCentered.txt";

String soundFileAddress = directory + "\\Sounds and Places\\OEDSounds Final.txt";

String chapterSoundsFileAddress = directory + "\\Sounds and Places\\ChapterSounds.txt";

String chapterSoundsSentencesFileAddress = directory + "\\Sounds and
Places\\ChapterSoundsSentences.txt";

String notPlaceFileAddress = directory + "\\Sounds and Places\\Not A Place.txt";

String chapterPlacesFileAddress = directory + "\\Sounds and Places\\ChapterPlaces.txt";

String chapterPlacesSentencesFileAddress = directory + "\\Sounds and
Places\\ChapterPlacesSentences.txt";

MesosticMakerLonger mml = new MesosticMakerLonger(mesosticRowFileAddress,
chapterFileAddress, allMesosticsFileAddress);

                mml.runMaker();


//MesosticMakerShorter mms = new MesosticMakerShorter(mesosticRowFileAddress,
chapterFileAddress, allMesosticsFileAddress);

                //mms.runMaker();


//MesosticFinisherAdjacentWordAdder mawa = new
MesosticFinisherAdjacentWordAdder(chapterFileAddress, allMesosticsFileAddress,
allMesosticsWithAdjacentFileAddress);

                //mawa.runMaker();


//MesosticFinisherCentered mfc = new
MesosticFinisherCentered(allMesosticsWithAdjacentFileAddress, allMesosticsFinalCenteredAddress);

                //mfc.runMaker();
```

```
//SoundGetter sg = new SoundGetter(chapterFileAddress, soundFileAddress,
chapterSoundsFileAddress, chapterSoundsSentencesFileAddress);

                    //sg.wordGetter();

                    //sg.sentenceGetter();


//PlaceGetter pg = new PlaceGetter(chapterFileAddress, notPlaceFileAddress,
chapterPlacesFileAddress, chapterPlacesSentencesFileAddress);

                    //pg.wordGetter();

                    //pg.sentenceGetter();


          }
}END
```

```
public class SyllableWebSearchTester {
public static void main(String[] args) throws InterruptedException, IOException {
// input test variables
String directory = "C:\\Users\\Martin\\Documents\\MSc Software\\Final Project\\Star Factory
Text\\Reference Documents\\";
String mesosticRowFileAddress = directory + "MesoticRow.txt";
String mesosticRow = new String(Files.readAllBytes(Paths.get(mesosticRowFileAddress))).trim();
String[] mesosticRowArray = mesosticRow.split("");
int mesosticRowArrayElement = 0;
String thisMesosticLetter = mesosticRowArray[mesosticRowArrayElement];
String word = "facTory";
/*
* Set a system property to "geckodriver.exe", the necessary engine for linking Selenium and Firefox.
 * create a Firefix WebDriver and get the target website
 *
 */
System.setProperty("webdriver.gecko.driver",
        "C:\\Users\\Martin\\Documents\\Java Libraries\\geckodriver-v0.17.0-win64\\geckodriver.exe");
        WebDriver driver = new FirefoxDriver();
        driver.get("http://www.howmanysyllables.com/");

        /*
         * Find the textbox WebElement for searching the dictionary and input
         * word test variable (note: could write this on one line without
         * creating a WebElement object:
         * driver.findElement(By.name("SearchQuery_FromUser")).input.sendKeys(
         * "factory");)
         */
        WebElement input = driver.findElement(By.name("SearchQuery_FromUser"));
        input.sendKeys(word);

        // Find the button to submit text to the dictionary and click it
        // (note could also write this on one line without creating a WebElement
        // object)
```

```
        WebElement submit = driver.findElement(By.id("SearchDictionary_Button"));
        submit.click();

        /*
         * Create a WebElement of the output from the website (test word divided
         * into syllables)
         *
         * (note about the website code: the className "Answer_Red" is used more
         * than once on the page, but only once within the
         * "SyllableContentContainer" element of the page, hence this nested use
         * of findElement operator)
         */
        WebElement result =
driver.findElement(By.id("SyllableContentContainer")).findElement(By.className("Answer_R
ed"));
        // divide the output into an array of syllables
        String[] wordAsSyllables = result.getText().split("-");
        // save and print the syllable that contains the target letter
        String savedSyllable = wordAsSyllables[0];
        for (int i = 0; i < wordAsSyllables.length; i++) {
        if (wordAsSyllables[i].contains(thisMesosticLetter)) {
                savedSyllable = wordAsSyllables[i].trim();
                }
        }
                // print the output as a String
                System.out.println("PRINT TEST");
                System.out.println("Word variable divided into syllables: " + result.getText());
                System.out.println("Syllable to save: " + savedSyllable);

                //close Firefix page and quit Firefox
                driver.close();
                driver.quit();
        }
}END
```

```
public class MesosticWordFinderTester {

public static void main(String[] args) throws IOException {

// initialize strings for source file paths (mesostic row and chapter)

String mesosticRowFileAddress = "C:\\Users\\Martin\\Documents\\MSc Software\\Final Project\\Star Factory Text\\Reference Documents\\MesoticRow.txt";

String chapterFileAddress = "C:\\Users\\Martin\\Documents\\MSc Software\\Final Project\\Star Factory Text\\Reference Documents\\01 RAGLAN STREET.txt";

// upload source files as Strings

String mesosticRow = new String(Files.readAllBytes(Paths.get(mesosticRowFileAddress)));

String chapterFile = new String(Files.readAllBytes(Paths.get(chapterFileAddress)));

// create a String array of the target letters

String[] mesosticRowArray = mesosticRow.split("");

String thisMesosticLetter = mesosticRowArray[0];

String nextMesosticLetter = mesosticRowArray[1];

// create String array of the ChapterFile

String[] chapterArray = chapterFile.split("\\s+");

// identify a word and letters to find in the array

int wordIndex = 138;

String word = chapterArray[wordIndex];

String wordIndexAsString = " ";// required for String[] output below

String letter = String.valueOf(word.charAt(0));

int letterIndex = 0;

String wordSegment = word.substring(letterIndex);


// test output

System.out.println("OUTPUT TEST 1");

System.out.println("wordIndex is: " + wordIndex);

System.out.println("wordIndexAsString is: " + String.valueOf(wordIndex));

System.out.println("word is: " + word);

System.out.println("Target This Letter is: " + thisMesosticLetter);

System.out.println("Target Next Letter is: " + nextMesosticLetter + "\n");


// find word that contains thisMesosticLetter

for (int i = 0 + wordIndex; i < chapterArray.length; i++) {
```

```
if (chapterArray[i].toLowerCase().contains(thisMesosticLetter)) {
		//assign wordIndex
		wordIndex = i;
		wordIndexAsString = String.valueOf(i);
		// assign word, change to lower case, and remove all non-words
		word = chapterArray[i].toLowerCase().replaceAll("\\W", "").trim();
		// output test
		System.out.println("OUTPUT TEST 2");
		System.out.println("wordIndex is: " + wordIndex);
		System.out.println("word is: " + word + "\n");
		// split word at thisMesosticLetter
		for (int j = 0; j < word.length(); j++) {
		letter = String.valueOf(word.charAt(j));
		if (letter.equals(thisMesosticLetter)) {
			letterIndex = word.indexOf(letter);
			wordSegment = word.substring(letterIndex);
			break;
		}			}

		// format word for mesostic output
		if (!wordSegment.contains(nextMesosticLetter)) {
			letter = letter.toUpperCase();
			word = word.substring(0, letterIndex)
			+ letter + word.substring(letterIndex + 1);
			//String[] output = { wordIndexAsString, word };
			// output test
			System.out.println("OUTPUT TEST 3");
			System.out.println("Letter is: " + letter);
			System.out.println("Index of Letter is: " + letterIndex);
			System.out.println("wordSegment is: " + wordSegment);
			System.out.println("Word with uppercase FoundLetter is: " + word + "\n";

			break;
			} else
				wordIndex += wordIndex ;
```

```
                    word = chapterArray[wordIndex];
                } else
                    word = chapterArray[i];
            }
            String[] output = { wordIndexAsString, word };
            // output test
            System.out.println("OUTPUT TEST 5B");
            System.out.println("wordIndex is: " + wordIndex);
            System.out.println("word is: " + word);
            System.out.println(output[0] + "\t" + output[1]+ "\n");
} END
```

Appendix 6. Sample Outputs from Tests on Owenvarragh Test Data

AllMesostics:

0	BEGIN
1	sTreet
14	wHo
16	sEated
22	outhouSe
25	The
26	Age
27	wheRe
45	oF
46	nAture
79	faCe
91	overTaken
95	Of
102	cRamped
111	memorY
113	narraTive
119	Him
121	cigarEtte
122	becomeS
127	iTs
128	Animated
131	dRaws
151	oF
154	punctuAte
162	tendenCy
164	gaTher
165	intO
166	oRnate
192	instantlY
194	iT
220	tHroughout
221	thE
226	alwayS
230	Telling
236	And
238	diffeRent
240	oF
241	emphAsis
242	eaCh
243	Time
244	rOund
247	cisteRn
290	privY
10000	END

AllMesosticsPlusAdjacent:

1 sTreet it is cold and dark and i am standing facing
14 my father wHo is
16 sEated on the throne of the
22 outhouSe i am
25 The
26 Age
27 wheRe our heads are level with each other i am there
45 because i did not want his call oF
46 nAture to interrupt the story hed been telling me so
79 cigaretteend as he draws on it illuminates his faCe sporadically brief looks of dialogue are
91 before we vanish again overTaken by the realm
95 Of his voice which extends beyond the
102 cRamped dimensions of the outhouse into the space of
111 memorY and
113 narraTive as the words unreel from
119 Him his
121 cigarEtte
122 becomeS a visual aid and
127 iTs
128 Animated lipstick blip
131 dRaws timelapse squiggles on the 3d blackboard dark
151 these imaginary pictures he makes curvy waves oF possibility which
154 punctuAte or illustrate the storys rhythm and its
162 tendenCy to
164 gaTher
165 intO
166 oRnate runs and turns i see it like some instantrecall
192 spirals of dna red neon the writing fades as instantlY as
194 iT is written but our tooslow brains retain its
220 the beginning or a middle section linger on tHroughout
221 thE predetermined narrative predetermined yet
226 alwayS new because each
230 Telling of the story is rehearsal
236 And gains
238 diffeRent subtleties
240 oF
241 emphAsis
242 eaCh
243 Time
244 rOund the
247 cisteRn whispering for example at some appropriate cold

AllMesosticsCentered

(Note: when pasted into this word format, the centring of the lines is incorrect)

1 sTreet it is cold and dark and i am standing facing
14 my father wHo is
16 sEated on the throne of the
22 outhouSe i am
25 The
26 Age
27 wheRe our heads are level with each other i am there
45 because i did not want his call oF
46 nAture to interrupt the story hed been telling me so
79 cigaretteend as he draws on it illuminates his faCe sporadically brief looks of dialogue are e
91 before we vanish again overTaken by the realm
95 Of his voice which extends beyond the
102 cRamped dimensions of the outhouse into the space of
111 memorY and
113 narraTive as the words unreel from
119 Him his
121 cigarEtte
122 becomeS a visual aid and
127 iTs
128 Animated lipstick blip
131 dRaws timelapse squiggles on the 3d blackboard dark
151 these imaginary pictures he makes curvy waves oF possibility which
154 punctuAte or illustrate the storys rhythm and its
162 tendenCy to
164 gaTher
165 intO
166 oRnate runs and turns i see it like some instantrecall
192 spirals of dna red neon the writing fades as instantlY as
194 iT is written but our tooslow brains retain its
220 the beginning or a middle section linger on tHroughout
221 thE predetermined narrative predetermined yet
226 alwayS new because each
230 Telling of the story is rehearsal
236 And gains
238 diffeRent subtleties
240 oF
241 emphAsis
242 eaCh
243 Time
244 rOund the
247 cisteRn whispering for examp