Algorithmic Music Composition Using Transitional Matrix Technique

UNIVERSITY OF LIVERPOOL

Benjamin Henshall

200965080

Dr. Igor Potapov

Contents

Abstract

Algorithmic composition of music has always tried to branch the idea of artificial “intelligence” with culture. Identifying what makes music produced sound aurally pleasing is challenging to build into a system such that using existing sources of music may be attractive.

The aim of this project is to develop and test the idea of a building a music probabilistic matrix and using it attempt to make original and interesting music. Using original classic compositions (such as Bach, Beethoven) the program builds a probability matrix composed of Markov chain probabilities. These probabilities are then used to generate a new, original piece of music using user oriented parameters.

The results of the project show that the problem is more complex than simply analysing the pitches of existing compositions. Instead, this project should be treating as a musical experiment of sorts, with the possibility of extending it further to allow more extensive testing of data.

Introduction

The project is aimed to investigate what the effects of generating music using a stochastic matrix produced by existing compositions. It should be viewed more as a musical experiment and can be used to evaluate how effective the use of stochastic matrices/Markov chains are when randomly generating music.

The project is intended to be user friendly and user oriented, so that users can easily change the parameters used by the system, having differing effects on the final score produced. Users will be able to give the program a set of some audio files and should receive a score that maintains some style of the imported files, but is still different in some ways. In addition, the user will be able to view a visualisation of the imported tracks in the form of a stochastic matrix, colour coordinated to display the likelihood of an event.

A large issue with the project, similarly to other music generation projects, is that the end product is hard to evaluate. When a project is designed to filter a database, is can easily be tested and verified to ensure the system is working completely as intended, since the main requirements of the system are functional requirements. My project, and many other projects in this domain, have the main requirement as a non-functional requirement – something that is difficult to test and is subjective. Another challenge I faced while undertaking this project is my lack of experience in Musicology. I listen to a large amount of varied music, but I do not have experience in music theory or fundamental musical concepts such as notation. I had to learn these throughout the project as I progressed.

The final solution I have produced heavily resembles the aims and objectives I set out to achieve, as well as closely matching the design. I have created a Java program with the ability to import MIDI files and build a stochastic matrix of note sequences. The program can then generate a new unique score that (occasionally) resembles parts of the imported compositions. Finally, the user can modify this newly generated scores or the rules by which it is generated, such as tempo, first notes and instrument. All of this functionality is presented to the user in a well formed Java Swing interface.

Overall, I'd call the project a success in one area, but a failure in another. If I were to evaluate the project over how well the project has run, how close the implementation is to the design as well as accurately the system represents my original aims, then I'd call the project a success. The area in which it fails, which is a large portion of the program, is the actual music generation. I would have liked to have the program produce aurally pleasing sounds more often than currently, but I fear that this is limited by the means I use to generate them, the stochastic matrix idea. The program merely acts upon the matrix and retrieves values. There are no real musical theories been implemented.

Background

Going as far back as 17th century (Algorithmic Composition: Paradigms of Automated Music Generation p36) we have seen musicians attempt to randomize the composition using techniques such as musical dice and ring composition [1] (Algorithmic Composition: Paradigms of Automated Music Generation p37). Like most methods now, even with much higher computational power, such approaches require analysis of the end product in order to verify the quality of the method. That is the largest difficulty with algorithmic composition, when compared to other tasks we perform computationally. For example, take an algorithm that compresses video or an algorithm that processes large sets of data. Both of these can be verified as working successfully and given values as to how well they perform their job (video quality, time taken, memory usage) whereas the end product of a musical form can only be verified by a human as it is subjective. This subjectivity also makes it difficult for us to implement music theory into the algorithm, as it is not an exact science.

A Markov chain style approach has been used previously by Harry F. Olson in the 1950s [2] as well as Iannis Xenakis through the 60s and 90s [3] but both lacked user customization and ease of use, focusing more on being able to use the piece that was finally produced. I would like my project to be more user friendly, allowing users with near no technological or maths background to be able to configure and create new scores.

To prepare for my project, I first researched about randomly generated numbers and how to manipulate the outcome of random functions, such as Gaussian functions, Cauchy functions and Weibull functions. I performed a small amount of research on composing music using mathematical sequences such as pink noise (1-over-f) or fractals. Since I decided to not use these sequences in my solution, I have declined to research further. The bulk of my research has been into using transitional tables, which were originally used in Olson in the 1950s. I have also followed many of the tutorials available on the jMusic website [4] including a score that produces notes in the form of a sine wave. I will continue to research about jMusic functions that may be useful to me and also about probabilistic automata to try and make the overall structure of the music more dynamic. I also refreshed myself with Java swing, a GUI library for Java, as I will likely build my interface in this.

The original requirements for the project was as follows:

The system must be able to:

- Translate existing pieces of music into a form of data that can be used for analysis by an algorithm and display it to the users.
- Generate an original score of music in a similar style to music given to the program
- Allow users to interact with the program through a well formed GUI

The system should be able to:

- Allow users to edit the data translated from the music given to the program in a GUI
- Allow the user to switch between different instruments, starting octave and beats per minute for the generated score

- Allow users to select how random they want the selection to be based on a scale in the GUI

Data Required

For the musical composition aspect of my programming, I used a library called jMusic, created by Andrew Sorensen and Andrew Brown. Information about jMusic can be found at http://explodingart.com/jmusic/.

I used digital MIDI files gathered from http://www.8notes.com/, which are royalty free recordings of public domain compositions.

I used no human data or human participants in my project.

Design

NOTE: Changes from the original design are highlighted in red. Comments and justifications on the changes are preceded by ***.

Summary of proposal

The aim of my project is to create a system that, given several composed pieces by a composer, will create an original piece of music that maintains the style of those fed into the system. This should include a way for users to manipulate the data translated from the pre-composed pieces in order to re-adjust the sound and improve upon it.

Compared to the original specification, I would like to focus more on the user interaction with the software, allowing them to readjust the randomness within the system to create original sounds, which maintain the style of the composer.

In preparation for this project, I have spent a large amount of time researching the functionality of jMusic, which I will be using to create the software. jMusic is a powerful Java library which allows a user to compose music using the Java language structure, allowing for deeper, richer logic and more efficient algorithms when it comes to creating the music. I have thoroughly tested jMusic and understand the functionality it has which includes the ability to construct transitional matrices from MIDI files which includes pitch values, rhythm values and dynamic values.

In addition, I have heavily researched the options I have for randomly generating numbers. Using Gaussian functions or Cauchy distributed numbers will help with when I decide how random I would like the system or how random the user would like the system.

Finally, I have decided to use Github to manage the project files. This will allow me to easily revert any important changes made to the program. ***I attempted using Github partway through the project, but had several issues with it and Eclipse. Instead, since I don't need it over several devices, I opted to leave Github for the time being.

Glossary

Due to jMusic not being a popular Java package, I have chosen to include a small glossary of terms for users to explain the terminology I use throughout the document.

- PitchValues, RhythmValues & DynamicValues:
  jMusic constants which can be referenced to as Strings or ints/doubles, i.e. PitchValue for pitch C of octave 4 = C4, which is equivalent to 60. Each Note I generate in my program has a PitchValue, RhythmValue and DynamicValue associated with it.

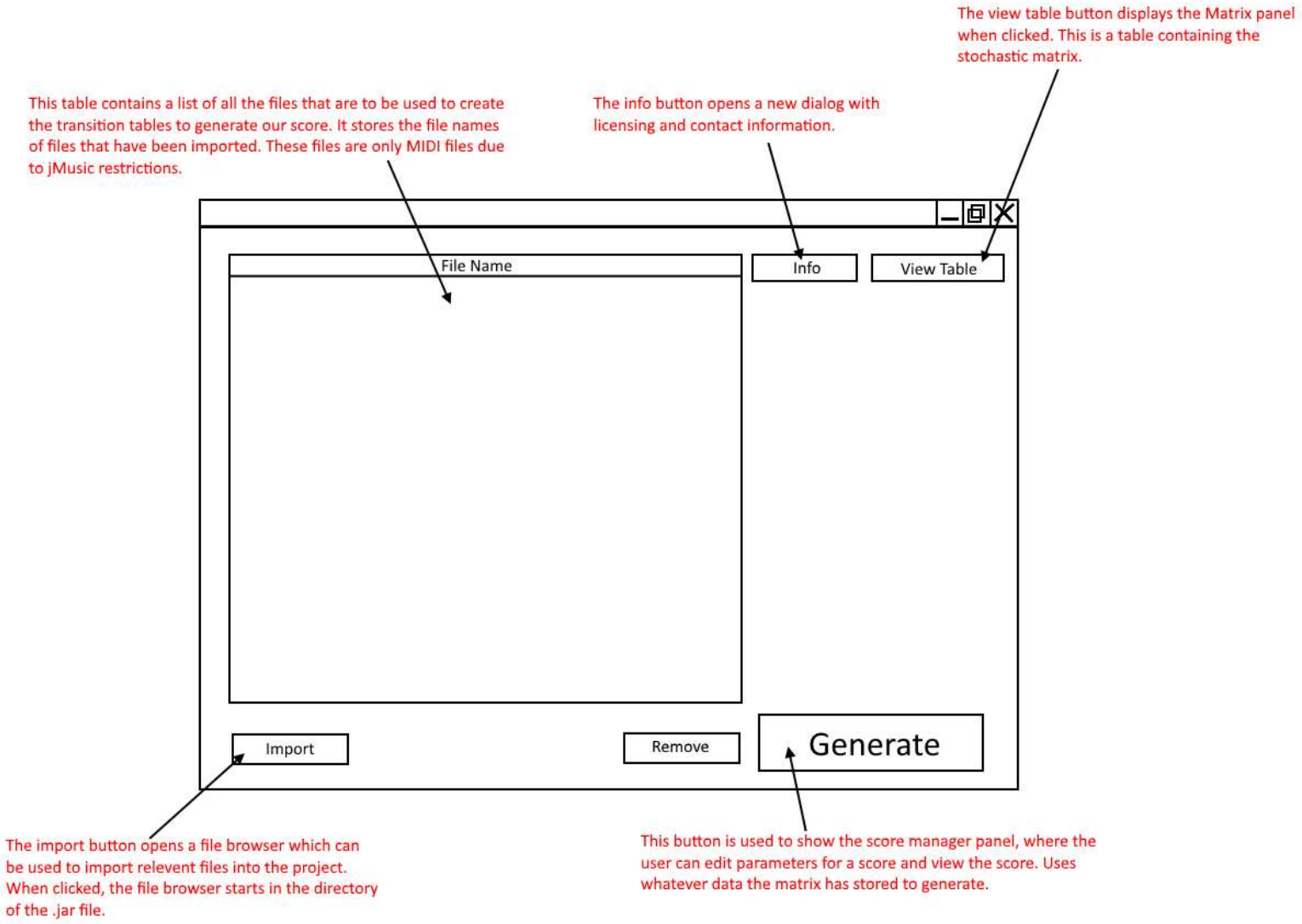
- Note
  jMusic object used to store details about a note. This includes a PitchValue, RhythmValue and a DynamicValue along with other values that I do not use. Multiple Notes can be added together into a part or a phrase

- Part
  jMusic object used to store a single instruments part. Usually made of phrases but I ignore Phrases as I do not need them. Many parts are often added together to create a score.

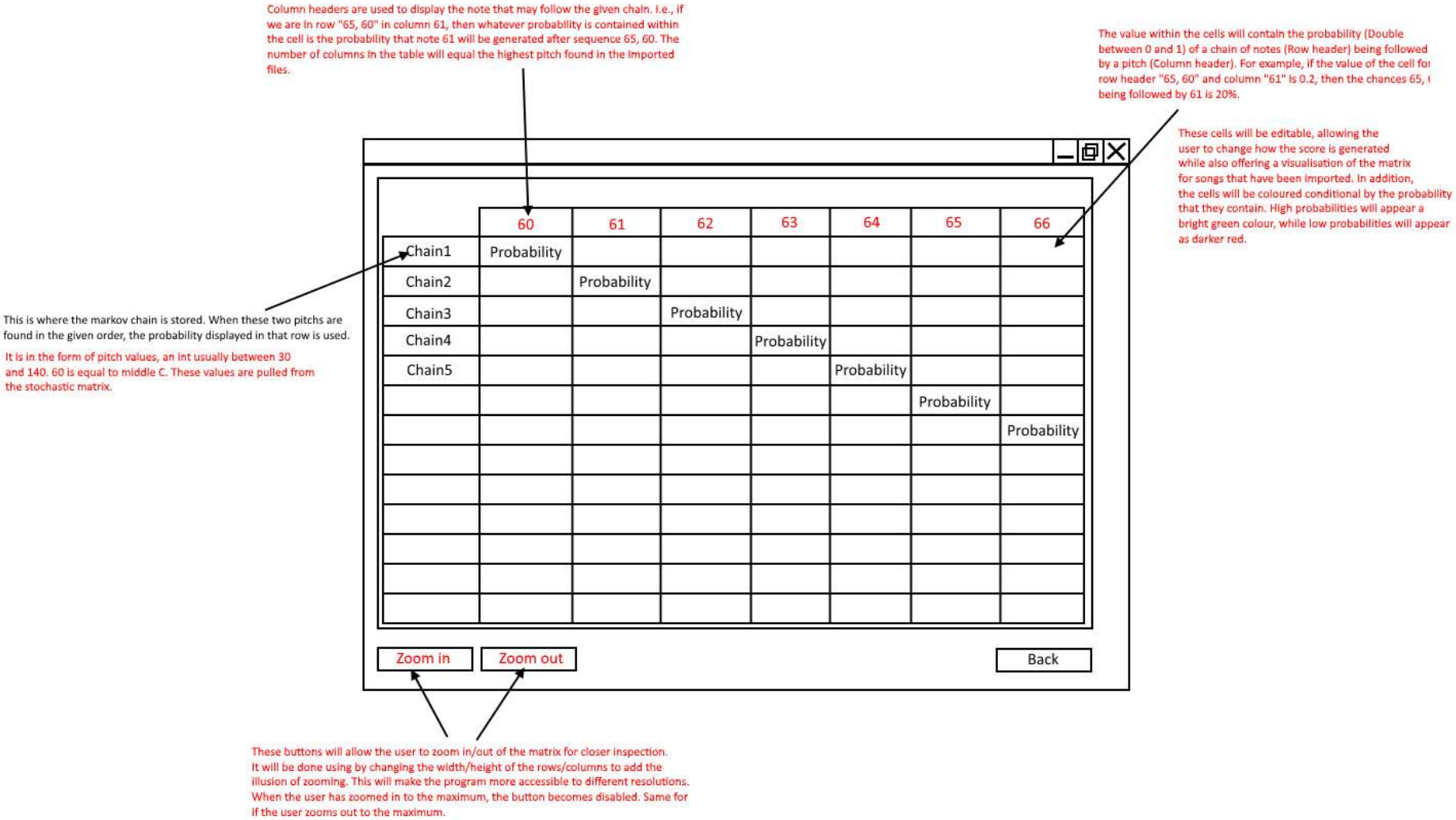
- Score
  A score is a jMusic object which contains one or more parts. It is representative of the final piece. When reading in MIDI files, I read them into a score and divide them up further if needed.

- Transition Matrix
  A transition matrix is a matrix that is used to store the probabilities of a Markov chain. It is what I use to generate the notes used in the score. I have chosen a transition matrix of depth two as bigger causes large inefficiency problems and smaller makes the score too random.

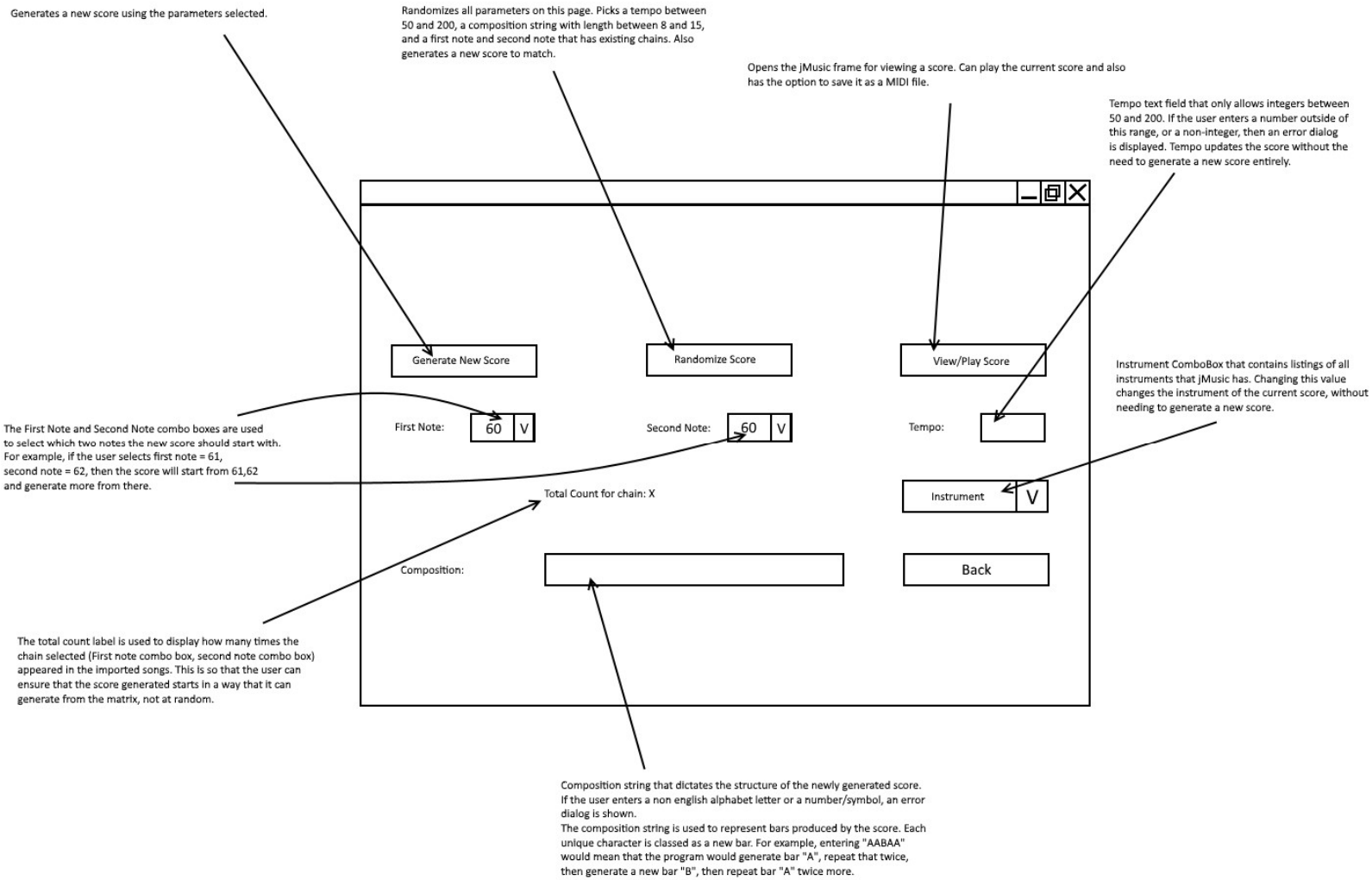
Main Interface view



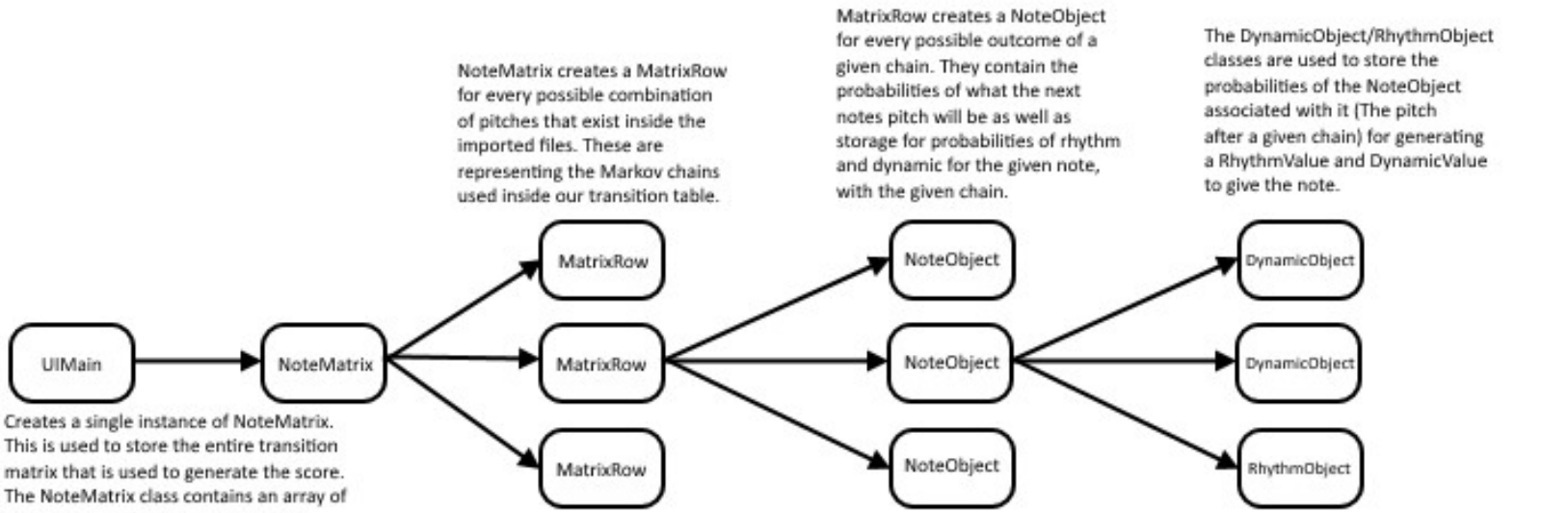
Transition View Matrix



Score Manager Panel



Visual of Matrix Layout



For a better idea of how my system works, here is an example:

The program uses MIDI (.MID) files to create the transition tables. These are imported into a jMusic format which contains a set of Notes. Notes can be broken down into three values, `PitchValue` (Pitch of the note), `RhythmValue` (Time interval of note) and `DynamicValue` (Loudness of note).

Importing into a jMusic format is done through using the read command to import into an array of Parts. Parts are split into phrases and then into Notes. This is an unfortunate functionality of jMusic so I cannot immediately grab the set of Notes from the MIDI file.

Once the MIDI file is imported and split, I will add them to

Due to the limitations of the built-in `TransitionMatrix` class in jMusic I have opted to manually create my own matrix. This enables me to be able to update the existing matrix and add multiple songs to it – the built in function only allows 1 track per matrix. I had considered stitching MIDI files together in order to pass this, but that would require extra work by the user and would have (slightly) incorrect information stored due to the intervals between songs.

We have a song in a MIDI file that is as follows:

	PitchValue	RhythmValue	DynamicValue
Note 1:	68	1	80
Note 2:	70	.5	50
Note 3:	69	.5	80
Note 4:	68	1	60
Note 5:	70	2	30

First, we create a `NoteMatrix`. We send it an array with these Notes. The lowest pitch in our song is 68, so that is our `minimumPitch`. The highest is 70, so that is our `maximumPitch`. We use these to create the following array elements, where we use `PitchValues` as indexes to represent a Markov Chain:

`MatrixRow[0][0]` is equivalent to `MatrixRow[68][68]` – We minus `minPitch` from the array index to allow us to start at index[0].

`MatrixRow[0][1]` is equivalent to `MatrixRow[68][69];`

`MatrixRow[0][2]` is equivalent to `MatrixRow[68][70];`

`MatrixRow[1][0]` is equivalent to `MatrixRow[69][68];`

`MatrixRow[1][1]` is equivalent to `MatrixRow[69][69]` and so on so forth.

These `MatrixRow` elements contain the possibilities that the chain might result in, such as:

`MatrixRow[0][2]`, equiv. to `MatrixRow`[68][70] has the possibility of returning a note with pitch 69, as in the original song we had a chain of pitches “68, 70, 69”. In the current example, this is a chance of 100% as there is no other “68, 70” chains.

**This idea has been removed. I now use a matrix where MatrixRow[0][0] represents pitch 0. The reason for the change is so that the user is still able to edit low probabilities even if they did not exist

in the imported songs. With the old system, if the user imported a song where the lowest pitch was 70, they would have no access to changing the pitch probabilities for, say, 65.

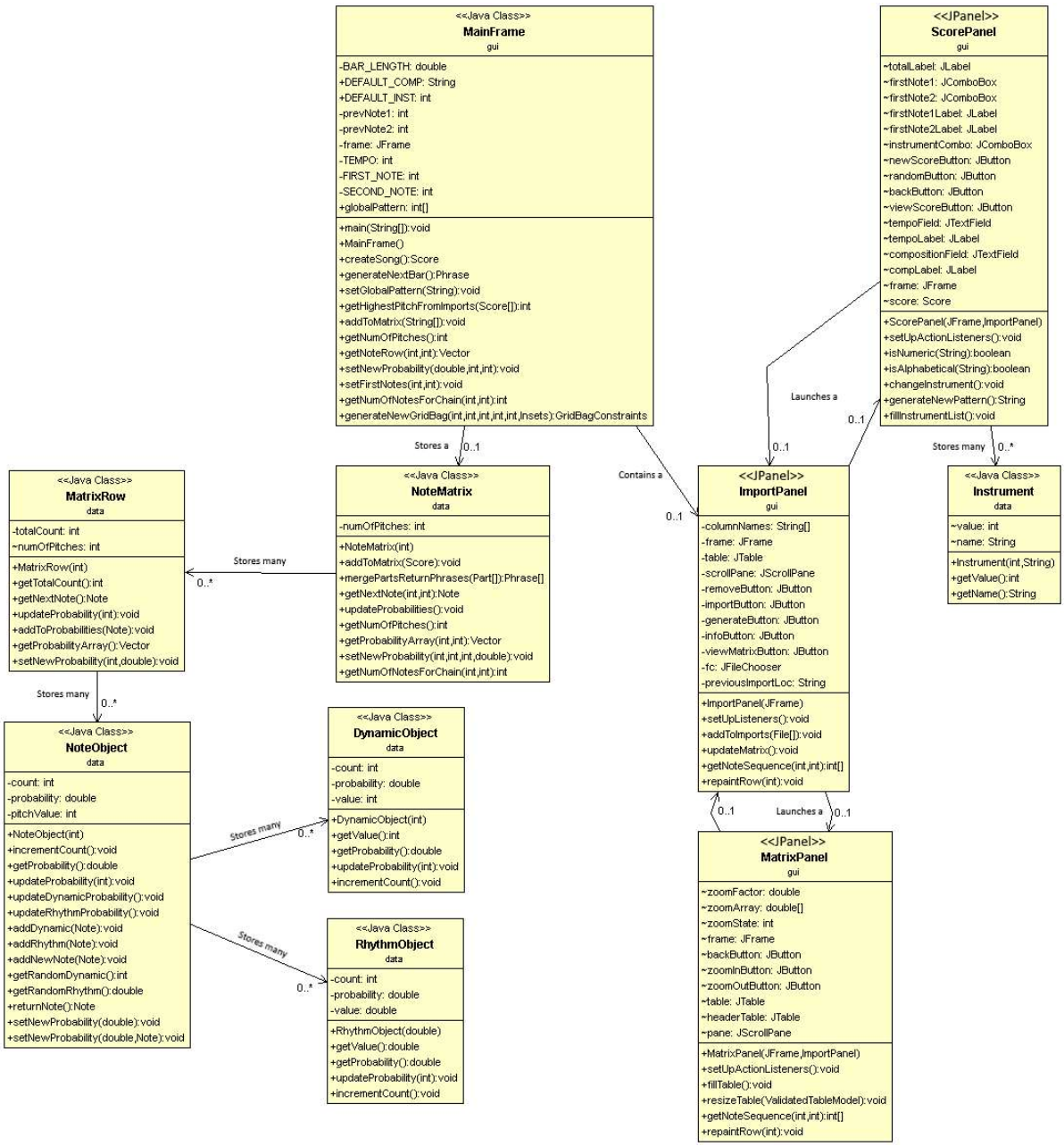
The `MatrixRow` element stores the probabilities in an array of `NoteObjects`.

`NoteObjects` store the probabilities of a pitch occurring as well as the probabilities of a rhythm or dynamic being selected IF that pitch is selected.

So the `NoteObject` storing data for the chain of “68, 70, 69” would have the probability of the dynamic being 80 as being 1 or 100%, as it is the only time that chain occurs.

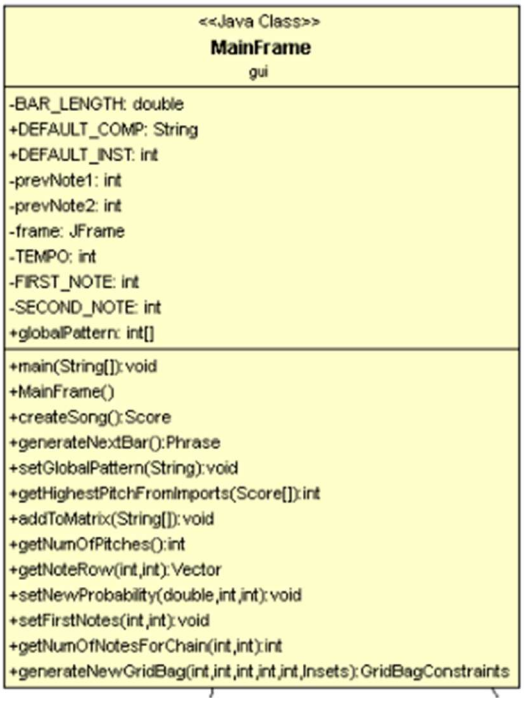
This program does not scale well with small amounts of source material due to the chances of `RhythmValues` and `DynamicValues` for chains being very high, but will scale very well with large amounts of information at its aid.

Component Design



Description of Components:

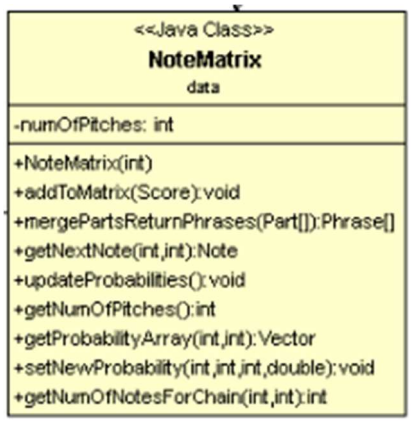
Data Components



MainFrame

The MainFrame class is the base of the program, connecting both the GUI and the matrix together. It has several utility methods such as generateNewGridBag and getNoteRow. These are static and are used by several classes. All fully capitalized variables are parameters that affect how the score is generated and can be changed through the ScorePanel.

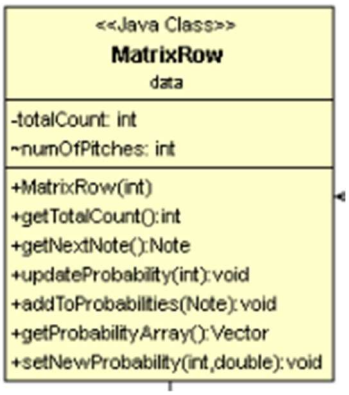
- CreateSong generates and returns a score using the parameters currently set.
- GenerateNextBar generates a bar, which is then added to the score during the CreateSong method call.
- setGlobalPattern is used by ScorePanel to set the composition string.
- getHighestPitchFromImports will be needed in order to know what size to make the matrix.
- addToMatrix will add a number of scores to the matrix, this will be used by the ImportPanel to add imported songs to the matrix (The string[] will contain file paths).
- getNumOfPitches returns the highest pitch found in the matrix, used when creating the matrix to decide how many columns are needed.
- getNoteRow returns a row of probabilities when given the note sequence, this is then used to fill out a row in the matrix table in MatrixPanel.
- setNewProbability is used to edit the probability of a note when given the sequence and a probability.
- setFirstNotes sets the first two notes to be used when generating a new score
- getNumOfNotesForChain returns the number of times the given chain was found in the imported songs
- generateNewGridBag returns a GridBagConstraints with the values passed to it, used as a utility class for the GUI to reduce code clutter.



NoteMatrix

Stores the stochastic matrix. Contains a 2D array of MatrixRows, which have an element for each possible chain of notes. The matrix is instantiated through the constructor, which creates it of size equal to the highest pitch found. The notes are added to the matrix through addToMatrix, which processes the score and adds the results to the matrix.

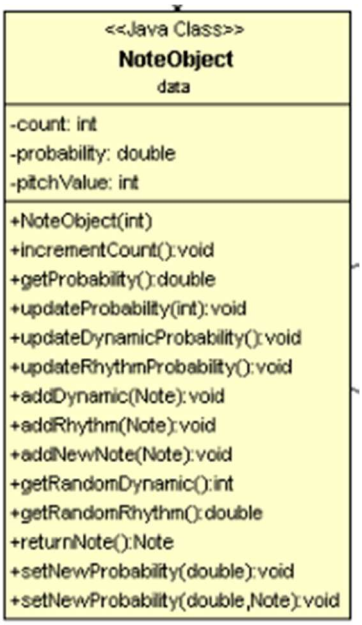
- mergePartsReturnPhrases will be used to combine a score into a single set of phrases for processing. This needs to be done due to some MIDI files being split into right hand/left hand parts.
- GetNextNote generates a note when given a chain of the two previous notes. Uses the probability stored in the MatrixRow[][] element that represents the given chain.
- getNumOfPitches returns the number of unique pitches in the matrix
- getProbabilityArray returns a note row
- setNewProbability edits the existing probability of a note event happening
- getNumOfNotesForChain returns the total count of notes found for a chain, by getting the count for MatrixRow[][] where the element represents the chain given.



MatrixRow

Stores the details for what is a row in the matrix. An instance of this class is created for every possible chain of notes. The count stores how many times the chain was found in the imported songs.

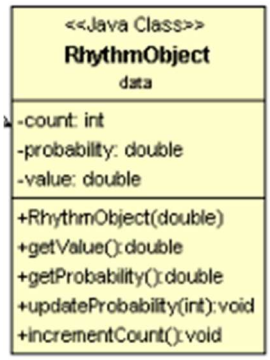
- getNextNote generates a random number and selects a note based on that random number (See pseudocode for more detail). Also calls to generate a new rhythm and dynamic then returns the note.
- updateProbability cycles through the array of NoteObjects and updates the probabilities to (count/totalCount)
- addToProbabilities adds a given note to the probabilities
- getProbabilityArray returns a vector containing the probabilities for a row in the matrix.
- setNewProbability is used to manually set the probability of an event.

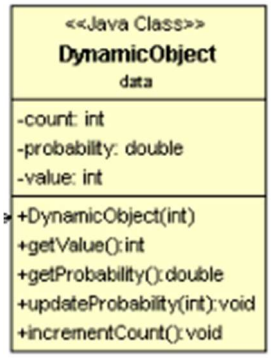


NoteObject

Stores the details about the event of a note being produced. Includes the probabilities, how many times this note was found for the given chain (Indicated by the hierarchy) and the actual pitch value of the note. Also includes an array of dynamics and rhythms used to generate a complete note.

- updateProbability calculates the probability using (count/MatrixRow.totalCount)
- updateDynamicProbability and updateRhythmProbability update the probability of rhythm/dynamic in the same way as updateProbability.
- addDynamic and addRhythm are internal methods used to add the dynamic/rhythm of the newly added note to the existing arrays
- addNewNote processes a note by adding the dynamic and rhythm to the arrays through the add methods and increments count
- getRandomDynamic/getRandomRhythm return a random rhythm/dynamic in the same way as getting random pitch (found in pseudocode).
- returnNote constructs a note using the pitch and a random dynamic/rhythm generated from the getRandom methods
- setNewProbability sets the probability of this pitch being produced to a new value. The overloaded method is used for notes that don't have a rhythm or dynamic.



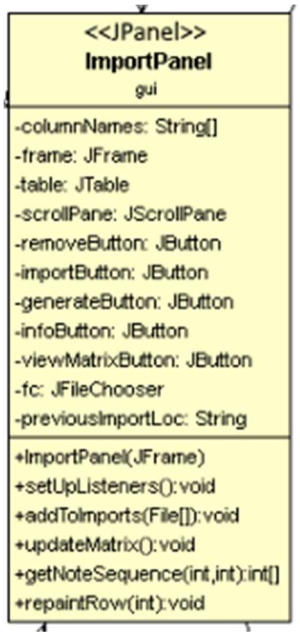


RhythmObject/DynamicObject

Both of these objects are exactly the same bar the value they store, as dynamic is an integer and rhythm is a double.

- getValue returns the value of the dynamic
- getProbability returns the probability of this dynamic occurring
- updateProbability updates the probability to (count/NoteObject.count)

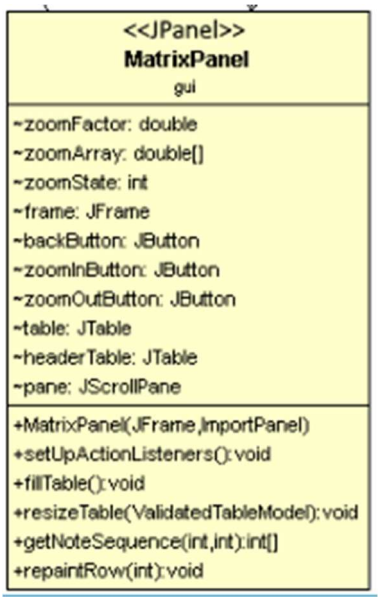
GUI Components



ImportPanel

Used to display files that have been imported as well as navigate to the matrix panel and score manager panel. Contains a table with the names of all the imported songs in. Uses a JFileChooser to select which files to add, including a string (previousImportLoc) to remember which directory the user was previous accessing.

- addToImports takes a list of files, which are converted to filePaths then sent to the matrix to be added as MIDI files.
- UpdateMatrix calls to update the probabilities of the whole matrix by refreshing their probabilities. Overwrites any changes made to the matrix through MatrixPanel.
- repaintRow repaints a single row of probabilities as it seems JTables do not have that capabilities. Refreshing an entire table with be too process heavy.



MatrixPanel

Used to display a table containing all the data from the stochastic matrix.

- zoomFactor and zoomArray are used to cycle through how far the user wants to zoom in. zoomArray will contain a list of possible values for multiplying column widths/row heights and zoomFactor will denote which is currently in use (zoomFactor of 1 means we're using the value in zoomArray[1]).
- fillTable fills the table using rows gathered from the matrix (getNoteRow in MainFrame)
- setUpActionListeners is simply a way of reducing code clutter by putting all action listeners in the body of code.

ScorePanel

Used to display all the parameters for editing how a score is generated. Leaves the actual score generating to the MainFrame class, but accesses MainFrame in order to get a new score. Parameters values are simply updated in MainFrame when changed.

- isNumeric and isAlphabetical methods will be needed in order to check if the composition string is a valid entry
- changeInstrument is used to update the instrument in use whenever we update the combo box or when we randomize.
- generateNewPattern generates a new random composition string and updates MainFrame

Key Methods

Method 1: `createTransitionTable`

The aim of this method is to create a transition matrix. The transition matrix is required to be able to generate notes and therefore create a score. The idea behind this method is that it creates a `NoteMatrix` that contains an array of `MatrixRow` arrays. We use a 2D array as it makes it easy to identify chains, by using the notation `[currentNote -2][currentNote – 1]`. Each `MatrixRow` contains a `NoteObject`, which stores the probability of creating the three elements of a note, the pitch, rhythm and dynamic. When we set these up, we also initialise count and probability to 0 – as there are no statistics in the structure yet so they can't be anything else. Once everything in `NoteMatrix` and its sub classes is initialised, we fill them with the data from the notes array we were passed in the constructor. Filling the data in is simply finding where this note belongs and then incrementing the count. Finally, we need to make sure the probabilities are up to date so we call a method to update the values for us.

```
Class: UIMain
Method: createTransitionTable()

INSIDE UIMain.createTransitionTable()
create new NoteMatrix with Notes[][] for imported files
    |
    |
    ------->INSIDE NoteMatrix CONSTRUCTOR(notes[] : Note[][]):
        set numOfPitches to number of unique pitchValues in notes[]
        for i=0 to numOfPitches
            for j=0 to numOfPitches
                set NoteMatrix[i][j] to new MatrixRow(numOfPitches)
                |
                |
                ------->INSIDE MatrixRow CONSTRUCTOR(numOfPitches : int, minPitch : int):
                |       for i=0 to numOfPitches
                |           set pitchProbabilities[i] to new NoteObject(minPitch+i)
                |           |
                |           |
                |           ------->INSIDE NoteObject CONSTRUCTOR(value : int):
                |           |       set pitchValue to value
                |           |       set count to 0
                |           |       set probability to 0
                |       End of loop
                |
            End of loop
        End of loop
        for i=2 to amount of notes in this array
            add notes[i] to matrixRow[pitchValue of notes[i-2][pitchValue of notes[i-1] using addToProbabilities method
            |
            |
            ------->INSIDE MatrixRow addToProbabilities(note : Note)
            |       for i=0 to pitchProbabilities length
            |           if pitch of pitchProbabilities[i] is equal to note pitch
            |               add new note to pitchProbabilities[i]
            |               |
            |               |
            |               ------->INSIDE NoteObject addNewNote(note : Note):
            |               |       increment count
            |               |       if DynamicValue of note exists in dynamicObject[]
            |               |           increment count of dynamicObject[i]
            |               |       else
            |               |           add new DynamicObject to dynamicObject
            |               |       if RhythmValue of note exists in rhythmObject[]
            |               |           increment count of rhythmObject[i]
            |               |       else
            |               |           add new rhythmObject to rhythmObject
            |       End of loop
            |
            |
        End of loop
        update probabilities
```

This will create everything needed to start generating notes. It ensures every chain and every scenario has a probability associated with it. It also ensures all the counts are set to the correct values, which also ensures that the probability is calculated correctly. It is worthy to note that each chain has a single pitch as the outcome and that pitch generates a single rhythm and dynamic that matches it.

Due to jMusic limitations, I am required to build a transitional table myself rather than using the built in function from jMusic due to not being able to use more than one source without the information being made incorrect.

Method 2: Generate part

The `generatePart` method is designed to create parts, using the `generateNote` method, for us to add to the score. We generate parts, instead of just adding the notes straight into the score, because we occasionally want to repeat sections to replicate a chorus. The method initialises which two notes to use in the Markov chain process then generates notes to add to the part until we've either gone over the allotted time for this part or we have hit the note limit specified by the user.

```
Class: UIMain
Method: generatePart() : Part

//variable used to ensure we only generate one section
set totalTime to 0

//Sets first two notes in part to the seed or the last two notes in the score
set seed[] to DisplayTransitionTable.seed[]
create new part[]
if score is null    //no parts have been made yet so add seed to beginning
    set part[0] to seed[0]
    set part[1] to seed[1]
Else                //part of the score exists so start where that finishes
    set part[0] to second to last note in score
    set part[1] to last note in score

//loop to produce a section that is ~16 measures long and not over the length bounds
while totalTime is less than 16 and totalNotes is less than number of notes to generate

    //generate next note
    create note and set to NoteMatrix next note method

    increment totalNotes

    //add to time counter
    increase totalTime by rhythm of note

    //add to part we will return
    add note to part

return part
```

This method is designed to create a part (A section) of the song. A single whole note is equal to 4.0 in the rhythm values of `jMusic` and we want a "part" to be 4 whole notes long which is equivalent to 16 measures. The method starts with either a seeded start or the previous two notes we generated. It then produces notes and adds them to the part until we have either passed our time limit or

passed our total note limit. It uses the `getNextNote()` method of `NoteMatrix` to generate new notes.

Method 3: Generate Note

This method generates all the notes that are eventually added to the score through `generatePart()`. It uses the probability in the transition matrix in order to generate a note that follows a similar form to that of the files that were input into the system.

This method, which is used heavily in the `generatePart()` method randomly generates a note when given a chain. The rhythm and dynamic are specific to that specific chain with that specific pitch outcome.

```
Class: NoteMatrix
Method: getNextPitch (pitch1 : int, pitch2 : int) : Note

return new note which equals get next note of matrixRow[pitch1-minPitch][pitch2-minPitch]
    |
    |
    --->INSIDE MatrixRow.getNextNote():
        generate random number between 0 and 1
        set ranTotal to 0
        for i=0 to numOfPitches
            add probability of pitchProbabilities[i] to ranTotal
            if random number <= ranTotal
                set note to pitchProbabilities[i].returnNote()
                |
                |
                --->INSIDE NoteObject.returnNote():
                |   create new Note
                |   get random dynamicValue from dynamicObject[] and add to Note
                |   get random rhythmValue from rhythmObject[] and add to Note
                |   add pitchValue to Note
                |   return Note
                |
        End of loop
        return note
```

The method generates a random number which is repeatedly compared to the random total. We continually added the probability to the total until we reach the random number set. This is a form of getting a random results from a set of probabilities.

Once the correct pitch has been found, we create a note inside `NoteObject` by getting the rhythm and dynamic (through the same method) and adding them, along with the `pitchValue`, to the newly created `Note`. Once the method is finished, we return the note back through the method calls.

Evaluation design

The main functionality of my program is to produce aurally pleasing music. This is by far the most important part of my program that will factor into decided if this project has been a success or not. It is hoped that the structure of the existing pieces that are fed into the program will have their effect maintained as the piece is scrambled and randomised through the transition matrix. Some of the effects that make the music sound good will be maintained, such as notes being closely grouped together rather than spread across the score or the effect of repetition of elements, which will be done when the program generates parts and decides when to re-use them.

Due to the subjective nature of music, it is hard to analyse whether a piece of music sounds good so I will attempt to make comparisons between my generated scores and the pre-composed pieces of music fed in. I will be looking for patterns such as:

- How many times is a section repeated?
  Repetition is an effective tool in music. Many songs consist of a repeated chorus and occasionally a repeated verse and it is well documented that repetition makes music, or any audio extract, more enjoyable to listen to.
  **Since composition structure is determined by the user, repetition is no longer randomized and this evaluation criterion does not matter.
- Is a note repeated more than twice?
  While repetition is good in music, repeating the same note more than a few times tends not to be. We want the sound to remain fresh and different, constantly changing and progressing the overall feel of the score.
- Did the score generate any noticeable patterns from the imported files?
  We can analyse how much of an effect a song has on a generated score by importing fewer files. Fewer files mean more chance for familiar patterns to appear and therefore less random.
- Are notes closely grouped enough? Are there any big gaps between notes that make the section sound out of place?
  When playing music, accidentally hitting a wrong note makes an instinctively “incorrect” sound, something that immediately sounds wrong. While this does still exist within small ranges, it is much more likely to happen over large ranges. In a short sequence of 5 notes, we shouldn’t be switching up two octaves. It makes the music sound “jumpy” or surprising. Surprising the listener is, most of the time, not wanted.
- Does the score maintain a relatively similar tempo or does it switch the tempo too much/too quickly?
  Similar to hitting keys that not within an octaves range, this can make the music sound too unstructured, too unnatural.
- Are there any patterns that are recognisable by viewing a histogram of the score and comparing it to pre composed pieces?
  Histograms give a good overview of the distribution of notes. If the values are too focused in certain places then the music might not have a huge amount of variety in it.
  **Removed this form of evaluation. Instead, I will listen for patterns by ear. The histogram functionality of jMusic is not very accurate and does not give enough information to evaluate from.

Likely the second most important function of my program is the ability to maintain the musical style of pre-composed pieces when creating a new randomly generated score. If this criterion is met, it will demonstrate the power of the transition matrix. It will be able to take in many different composed pieces from a single style, which could be a composer, a certain time period or even just a genre and be able to produce something interesting and comparable to the pre composed pieces.

This, however, will also be hard to measure. I will need to rely on using the patterns given to me by jMusic (In the format of the overall transition matrix table, a histogram or chord analysis of phrases) in order to identify how the composed pieces and newly generated pieces are similar. The best way to test this would be to feed the program a specific style of music that is very distinct, such as very high tempo, "happy" music. Something that has a very unique style and unique note patterns compared to normal composed pieces. I would not be able to recognise the difference between a score generating using Mozart compared to a score generated using Beethoven.

Those are the two main requirements of my project so should I deem either of them as incomplete, I will consider this project a failure.

In terms of how to format the evaluation, I will use a table containing several attempts at generating music under different parameters and rate them by each of the criteria. The parameters will be how many files imported, how good the new score sounds and how often patterns from imported files are found.

If these two functionalities work fine, however, the program should be evaluated on the following quantifiable measures:

- Do all elements of the GUI work as intended – Test each combination of options active/buttons being pressed.
- Does the randomisation scale make the music more/less random as intended – I could test this by looking at the jMusic sheet view and generating several pieces with low randomisation, then a few pieces with high randomisation and making comparisons between the two. The highly random pieces will likely have notes all over the sheet whereas will have occasional sections where the scale descends in a fashion similar to the original compositions fed to it. Alternatively, if the transition matrix has been thoroughly tested, I can view the transition matrix in its table view and compare the probabilities of low randomisation compared to high randomisation.
  **The randomisation scale was omitted from the program due to the complexity associated with including it, therefore this criteria is not used any more.
- Is the generated transitional table correct? – This can be tested with smaller composed pieces and results of method compared with my personal pen-and-paper calculations.
- Does score output to MIDI file correctly? – I can easily test this by checking the file system to see if a new file has been created. To check if the file is the same as the score generated by my program, I can read the score back into jMusic and compare the NoteValues to a printout from the generation program it was supposedly generated from.
- Does the score comprise of the correct amount of notes? – Simply add a statement to count the amount of notes that has been created and added.

Throughout the project I will adapt a rapid prototype style of testing, where I will continually build my system, add functionality, re-test, add functionality, and re-test until the entire system has been tested. I will be using blackbox test for this.

Realisation

Matrix:

I began the implementation stage of my project by building the Note Matrix that stores the probability of all chains of notes. This was a conscious decision as a large amount of the GUI code requires a source of data in order to display properly, for example, the Note Matrix table in the GUI would have no data to use a source and would therefore cause me to have to (essentially) develop the table twice, once for developing how it looks visually, but then again once I wanted to manipulate the data. I would have also needed to develop some test data to use which would have wasted even more time. I began by developing methods to build up a matrix of pitches, as adding the rhythm/dynamic probabilities to the system afterwards would not be too difficult. Due to the lack of visual representation of the data, however, verifying that the resulting matrix was correct was difficult. I used a large amount of print statements and Eclipse breakpoints to check the probabilities for certain pitches. Due to the structure of the data (A value inside an array inside a 2D array), finding the correct values took 2-3 minutes per time. The first two classes I implemented were the following:

NoteObject:

This class is used to store a pitch and the probability of that pitch occurring. It also contains the probability values for generating a dynamic and rhythm associated with that pitch. This class and its functions has remained generally the same as the design.

The only addition is adding a new method that is used for editing the probability through the GUI matrix table. I had originally planned to update a notes probability by simply adding to the count. In the implementation stage, however, I realised that this would not work, as continually increasing the total count of a row would cause increasingly large numbers to the point of breaking the system. Instead, I decided that I would only calculate probability by using (`count/totalCount`) when first building the matrix, then simply update the probability variable directly. This increases the simplicity of the code when updating the probabilities of an entire row in the matrix and increases the accuracy of such changes. I found the best way to implement this change was two separate methods for updating a probability. One for when the user tries to update the probability of a chain that never occurred in the imported songs and one for chains that HAVE appeared.

A chain that has never occurred in the imported songs will have no dynamic or rhythm attached to it, so it needs to be dealt with differently than one where a chain was found. I implemented this by having an overload method that took a note as well as the new probability, rather than just the probability. This new overloaded has to instantiate the dynamic/rhythm arrays too since they likely don’t exist. The values I add to these new arrays are just average values defined by me.

```
239     /*
240      * Method for setting a new probability value manually
241      */
242     public void setNewProbability(double newProb) {
243         probability = newProb;
244         // Rounds the value up for displaying on our table
245         probability = Math.round(probability * 100000.0) / 100000.0;
246     }
247
248     /*
249      * Another method for setting probability manually. This version is used to
250      * adds a dynamic and rhythm to pitches that have a count of 0, which
251      * wouldn't normally have a rhythm or dynamic but still need to generate
252      * one.
253      */
254     public void setNewProbability(double newProb, Note note) {
255         // If we haven't already created the array of dynamic/rhythm value
256         // probabilities, instantiate them
257         if (dynamicArray == null || rhythmArray == null) {
258             dynamicArray = new ArrayList<DynamicObject>(1);
259             rhythmArray = new ArrayList<RhythmObject>(1);
260         }
261         probability = newProb;
262         addRhythm(note);
263         addDynamic(note);
264     }
```

As I began building the MatrixRow class, I realised how having the matrix start from the lowest pitch found would be restricting to the user editing the matrix. If they wanted to manipulate the matrix to produce a much lower sound, they wouldn't have access to the lower pitches in the table due to pitches (usually below 40) simply would not be present in the matrix. Instead, I made the decision to have the matrix store all data from pitch value 0 rather than from the lowest found pitch. This also helped simplify the building of the matrix, as I could have the array indexes exactly match the pitch that was being referred to, such as if we added to pitch 68, we could easily reference pitch[68].

MatrixRow:

All of the methods that existed in my design for this class worked very similar to how I expected them to, with minimal changes. I did have to add two new methods, however. Similar to the NoteObject class, when planning the design of my system, I did not pay enough attention to how users would be interacting with visual representation of the matrix or how I would be filling it. I did not include a method for retrieving rows of data from the matrix, so I had to design it partway through the implementation process. In order to retrieve an entire row of probabilities at once, I repeated a loop once for each column in the matrix (equal to number of pitches) and added the probability for that column to the vector, before finally returning the vector for use in the visual table. This results in a vector where the first element is the probability of the chain generating a pitch of 0 and the last element in the vector represents the probability of the chain generating a pitch equal to the highest pitch from imported files.

In addition, I was required to design a new method with the purpose of updating an entire rows probability values. This is in use when the user edits a probability. I struggled to find a formula for this anyway, but managed to figure out a workable version on my own:

In order to get the probability correctly distributed when the user edits I had to create a custom formula. Since the total probability over a row must equal 1, I wanted the user to be able to edit a single probability and have the program automatically alter the other probabilities to compensate for the increase/reduction by adjusted the probabilities of other notes. For example, if I had a probability and 0.75 and one at 0.25 and I increased the 0.25 probability to 0.5, then the other

probability should reduce to 0.5 to keep the row value at 1. The important part of this, is keeping the ratio correct.

I developed the following formula to calculate this: With an example, where A, B, C are the only probabilities in a row, A' is the new probability:

A-K=A', so K is the difference between old and new probability

The new probability for B would be:

(K*B)/(B+C)+B

And the new probability for C would be:

(K*C)/(B+C)+C

So, for an example with actual values, imagine we have the row:

A B C

0.25 0.25 0.5

And if we edit A to make it 0.5 from 0.25, that gives us K=0.25

So the new probability for B would be:

(-0.25*0.25)/(0.25+0.5)+0.25 = .1666

And the new probability for C would be:

(-0.25*0.5)/(0.25+0.5)+0.5 = 0.333

These two new probabilities added together equal 0.5, which, when added

back to the new probability of A, equals 1.

Using this formula ensures that each rows summed up probability comes to exactly 1.0, while also maintaining the ratio of the probabilities that existed before the change. This works successfully in the program by getting the sum of all probabilities in the row, excluding the probability being edited, then performing the formula.

NoteObject

A large amount of changes were made to this class, mainly to make my code more readable. I realised that I no longer needed a method to return the pitch value or the count of NoteObject, as any processing involving those variables are done in NoteObject or lower in the hierarchy (Dynamic/Rhythm objects). I never need to know how many times a pitch has appeared following a

certain chain or the actual pitch value from within MatrixRow. Rather than having all the code to generate a rhythm and a dynamic inside the returnNewNote method, I split them into the new methods getNewRandomDynamic and getNewRandomRhythm, increasing readability and making it more maintainable. In a similar change, I created two new methods (addDynamic and addRhythm) that take a note and add the rhythm/dynamic of that note to the array. This, again, increases readability and maintainability, but also increases code reuse, as this method is used when setting a new probability for a pitch that has a count of 0.

As in MatrixRow, I had to implement a method that manually edits the probability variable, instead of probability being calculated by `count/totalCount` for the row. Both of these are simply setters that also round the probability to 6 decimal places and (in the overloaded method case) checks to see if we need to instantiate the rhythm/dynamic arrays.

RhythmObject/DynamicObject

Both of these are near exactly the same. The only difference being the value being stored for rhythm is a double, whereas it is an int for dynamic. I did face an issue related to this during implementation, as in the update probability function (Where it calculates probability as count/totalCount) I was not casting the count or totalCount to doubles, instead having the equation using them both as integers. This causes in issue in Java, where an int divided by an int always equals an int, meaning the probability was getting rounded down to 0. Having all probabilities at 0 caused my matrix to continually produce notes of dynamic 50 and rhythm 0.5, rather than having a variable value affected by the imports.

Once these classes were fully implemented, I progressed to developing the outer class that would store the chain that a row is stored in:

NoteMatrix

In the design stage of the project, I failed to realise how many methods for the matrix table I would need to add to the data structure of my program, in order to allow users to successfully view/edit the matrix on a GUI. I originally planned to build the entire matrix up through passing all of the notes of imported songs through the constructor and building it there. Instead, I opted for instantiating the 2D array of MatrixRow objects in the constructor, but then building up the probabilities and counts of the matrix through a new addToMatrix method. This keeps all dealing of data in the matrix class, rather than having the NoteMatrixs owner class having to split imported scores into parts, then into phrases, into notes etc. This increases maintainability and mobility of code. During implementation I failed to discover a large issue with my system and adding to the matrix. The MIDI file I was using was partly corrupt, meaning the pitches for some notes pitches were being valued at the overflow value for an integer. This went unnoticed for a while, as I believed it was an error with jMusic. Having the pitch for some notes be so high caused the program to create massive array of data, somewhere near to 1.5 million objects. This also caused the probabilities used while generating music to be completely incorrect, and meant that the generated score was almost 100% randomized, with no bearing to the imported songs. Luckily, I eventually caught it by trying another MIDI file and comparing. To fix the issue, I deployed a solution that checks that the pitch is higher than 0 before trying to add it to the matrix.

Due to my lack of experience with jMusic when designing the project, I thought that I would be able to cycle through the notes of a score without dividing the score up into a part array, then a phrase array, then a note array. Because this was not the case, I introduced a new method that merged all parts of a part array, then returned the phrase array of the newly merged part. This was done in a loop that uses a jMusic method called Mod.merge:

```
78      /*
79       * Method that merges all parts of a song together and returns an array of
80       * phrases. Most MIDI files are in separate parts, such as right hand/left
81       * hand, so this stage is necessary.
82       */
83      public Phrase[] mergePartsReturnPhrases(Part[] parts) {
84          // Cycles through the array of parts and merges them all into parts[0]
85          // using a jMusic function
86          for (int i = 1; i < parts.length; i++) {
87              Mod.merge(parts[0], parts[i]);
88          }
89
90          // Returns the array of phrases from the merged parts[0]
91          Phrase[] phr = parts[0].getPhraseArray();
92
93          return phr;
94      }
```

As mentioned before, the indexes of the 2D array had to be changed to start at pitch 0, rather than matrixRow[0][0] being equivalent to something like the chain 68, 68. getNumOfPitches is a getter added simply so the matrix table would know how many rows/columns the table would need.

Since the NoteMatrix controls the 2D array instances of matrixRows, any data that the matrix table needs has to go through methods in NoteMatrix. These are essentially wrapper methods that simply make a method call to the correct matrixRow. getProbabilityArray, setNewProbability and getNumOfNotesForChain are all like this, where they ask for data from the relevant matrixRow. getNumOfNotesForChain is used in the score manager panel to indicate how many times a chain occurred in the imported songs, so the user can see how random the beginning of the song is (i.e. if there were no chains occurred, it will be random for a few notes at the start of the newly generated score).

With the program able to build a full matrix, it was time to implement a class that could produce music using the probability matrix. To do this, I began to build a class containing the logic used to generate notes using the probability and also to decide upon the structure/composition of the music. In my original design, I included only two JPanel classes; one for the matrix and one for importing and editing score options. Instead of using this system, I made the choice to split it further into three JPanels, so that the score parameters (tempo, instrument etc) is on a third JPanel as to increase code maintainability and reusability. In addition, the design originally had the matrix contained within the JPanel class. I decided instead to add a “MainFrame” class that acts as a controller between the model (The matrix) and the view (The GUI classes). Doing so makes the code more modular and ensures classes don’t have conflicting purposes.

MainFrame

Probably the largest class in my program. Originally most of these functions were in the UIMain class, but I instead cleaned up the design and shifted the MainFrame to only have data processing

methods in, with the exception of creating a JFrame for putting panels in. The MainFrame is the starting point, containing a main method that only calls the constructor of its own class. Inside the constructor, a frame is created with an ImportPanel JPanel inside of it. All of the other methods and functionality are called as utility methods by the GUI elements. For this reason, all methods inside of MainFrame are static, as there is only a single instance available to use and is used by several classes. In addition, this class stores several constants that as global variables throughout the program, such as BAR_LENGTH which how many musical measures a bar should take up and the DEFAULT_INST which is the instrument that the program uses by default.

The first methods I created, which was before I had decided to use this as a class separate from the UI, were createSong and generateNextBar. createSong calls generateNextBar X number of times, where X is equal to the length of the composition string. Before I had thought of the idea of the composition string, I used a constant to decide how many bars that would be generated for the song.

I did not specify exactly how generating a song would work in my design (Only how notes would be generated) so I will explain in detail here. It includes the use of a counter that tracks how long the bar currently is.

1.) Generate a random note using the two previous notes in our score
2.) If we've already generated 75% of the bar, reduce the length of the note to end when the bar ends, update the previous notes to reflect us adding the new note, then add the note to the bar and update the time counter.
3.) Else if there is enough time in the bar for the new note generated to fit in, add the note to the bar and update previous notes.

This leaves us with a bar with a length equal to the constant BAR_LENGTH that we have defined. I will note that, due to each bar lasting an exact length, that there are no silent gaps in the produced song so it sounds odd. This is an intentional design choice, as when notes/bars overlap each other, they do not sound attractive in several instruments such as wind based instruments and piano.

The createSong method repeatedly calls generateNextBar until we have generated a bar for every phrase in the composition string. The idea of a composition string came to me close to the end of the project, while I was designing the final JPanel. The composition string is a user entered string that represents the structure of phrases for the newly produced song. Each unique letter counts as a unique phrase. For example, if the user enters the composition string "AABAAC".

The program would generate 3 different phrases, say if phrase A generated the notes "A, B, C", phrase B would generate another bar using the notes "B, C", as those are the two notes that preceded it. Phrase C would also generate using the notes "B,C" as those two notes preceded it. If Phrase B generated "D, B" and Phrase C generated "A, C" then this would make the final composition be "(A,B,C), (A,B,C), (D,B), (A,B,C), (A,B,C), (A,C)". Since repeated patterns in music increases the effectiveness of the music, this composition extra increases the chance of the music sounding good.

The composition string is given to the MainFrame class from the ScoreManager JPanel and is put into the globalPattern array by changing the string into a char array. This allows us to do a simple subtraction of the alphabets place in the ASCII table in order to have "A" equal to 0 and "Z" equal to 26. When createSong goes to use a bar, it first checks to see if that next bar being generated has already being generated and if it hasn't, it generates a new bar. For an example of composition string = "AACD", it first checks to see if phrase A is null, which it is, so it generates it. Then it moves

onto the second character in the string, which is also A. It checks if A is null, but since it isn't, it simply grabs that phrase from our phraseArray and reuses it. Both C and D are also null, so it generates them both as well.

Once a phrase is generated or copied from one already generated, it needs to have the time adjusted so that it starts in the correct place. It is then added to a part, which is finally added to the score when the part contains all the phrases.

As mentioned previously, there are a large amount of methods in this class that I did not design due to a different class structure as well as not knowing exactly how the matrix JTable would work. Most of these methods were designed/implemented during the building of the GUI, but I thought I'd explain them here instead.

Adding to the matrix is done through the addToMatrix method, which takes a string array of file paths. It reads the scores in by iterating over the array and using the jMusic Read.midi method then adding them to an array. We then instantiate the noteMatrix using the highest pitch from imported files. We can then iterate over the array filled with scores and add them to the matrix, as well as updating the probabilities to ensure they are value correctly.

getNoteRow is a wrapper method used to retrieve and matrixRow when given a chain. setNewProbability is also a wrapper, with the exception that it also has some logic to retrieve exactly which chain is being specified. The final wrapper is getNumOfNotesForChain, which retrieves how many occurrences exist for a given chain.

setFirstNotes is used when the user changes which two notes the new score should start with. It changes the parameters by which createSong uses.

The final method, which was only implemented during the refactoring phase of my project, after all of the functional code had been done, was generateNewGridBag. This is a simple method that generates a new GridBagConstraints class with custom variables sent through as parameters. This is less efficient that just declaring a new GridBagConstraints then manually changing the variables each time I want to add something to the GUI, but that means I have to use 4-5 lines for every component I want to add, rather than just the one. I made this change during refactoring in order to make the code more readable and more maintainable.

Once this method that actually produced a piece of music was somewhat finished, I moved on to creating the GUI for my program. I began by creating a new JPanel class called ImportPanel. This panel is used to import songs and add them to the matrix. This panel includes some functionality of the UIMain from my design, but with some methods omitted which were moved to the new ScorePanel class.

ImportPanel

True to my design, I kept the interface my simplistic. I have had past experiences with Java Swing and know how difficult it can be to get an interface that looks good. I accepted this and tried to focus on all the functionality being complete instead.

This panel has a few differences from the design, due to difficulties when implementing. I removed the "Length" and "Format" columns from the table that stores imported songs. I could not find any kind of library (Since jMusic does not have the functionality) that could extract meta data from a MIDI file, so I could not display the length of a track. I also discovered that jMusic functionality of

importing MP3 files is not very accurate, so I omitted that from the table too, leaving only the name of the file in the table. I also removed the idea of a pitch randomness slider, as I proved to be too difficult to implement. Evening out the matrix through a slider would have a massive amount of computation associated with it, as each change of the slider would have to calculate a new probability for `(highest pitch)^3` cells. A notable issue I had was with the table still being editable, rather than locked. I discovered through search engines that there is no build in function for this in a JTable/Table model. Instead, I had to implement my own class "UneditableDTM" which is an extension of DefaultTableModel but overrode the isCellEditable to always return false, rendered the table un-editable.

A new button, removeButton, was added towards the end of the implementation stage so that users could remove imported songs and update the matrix.

In order to clean up my code and make it more readable and more maintainable, I introduced a new method (I have similar methods in all UI classes) called setUpListeners. This method creates action listeners, including all the logic associated with them and adds them to the buttons they belong too.

Previously, I had assumed the program would need to import only a single file at the time due to my lack of knowledge with the JFileChooser object, so I only had a method for importing a single file at a time. Instead of this, I created the method addToImport(File[]). This takes an array of files, gathered through the JFileChooser and loops through them. If the file is not a MIDI file, it displays an error message and stops importing. It then checks the table to see if the file selected has already been imported, in which case it skips the current file. If not, it adds the name of the file to the table. Once all these files have been added to the table, the method calls to update the matrix. This gets the file paths stored in a hidden column of the import table and puts them into an array, which is then sent to the matrix for processing.

The getNoteSequence and repaintRow methods are wrapper methods, used to access the matrixPanel as it belongs to ImportPanel.

Note that originally my design (and earliest implementation efforts) had the generateScore button simply show the jMusic score frame, rather than an extra panel with more configuration details on.

As I now had the import interface correctly, I moved onto what was likely the hardest area of my project, the visual table displaying the matrix. This was mainly down to how difficult the JTable and DefaultTableModel are to use.

MatrixPanel

In my original plan, I had three tabbed panes in DisplayTransitionTable (Now named MatrixPanel), one for displaying rhythm, one for dynamic and one for the pitch. Partway the development of the matrix, I realised that this would not be possible. Imagine if we want to look at the rhythm probability for the pitch 86 following the chain 84, 84. We would go along to the row 84, 84 and to column 86. The dynamic and rhythm are kept as another matrix, meaning there wouldn't be a good way of displaying the matrix from inside a cell. An option I thought about was a small table to the right of the table that displayed the matrix for the rhythm/dynamic, but this would take up too much space as I want the focus to be on the table displaying pitches. Instead of this, I decided to keep the dynamic and rhythm un-editable by the user, although if I were to repeat the project and use a different method of storing my matrix (See Evaluation),I would figure out a better way of displaying this large amount of information.

So instead the MatrixPanel displays only the matrix for pitches. I began by trying to create an empty table with the correct number of rows and columns. In the first implementation of the MatrixPanel, I did not have a header table (mentioned later) and instead ignored the row headers completely, so it was hard to tell how accurate the information being displayed was. Producing a table with enough columns was fairly easy, simply a loop that adds a new column each time.

With an empty table, I moved to filling up the table with data. This is where the fillTable method was created. In order to get the probabilities back, I had to develop some methods (getNoteRow in MainFrame) to grab data from the matrix when given a chain. This method would be used n^2 times, where n is the highest pitch found. This is done through two nested loops, for example, if we have the highest pitch equal to 70, we would create the row for 0,0 then 0,1 then 0,2 up to 0,70 where we would then create the row for 1,0 then 1,1 etc until 70,70. I created the row as a vector as this easily fits in with the datatype of a DefaultTableModel row and can be added straight as a new row in the table.

As mentioned before, at this stage I had no headers for the rows so no indication of which row belonged to which chain. To solve this, I introduced a header table that is added to the scroll pane as a row header. Every time a row is added to the table, a new row with the corresponding chain is added to the header table. When this is added to the scroll pane as a row header, it ensures the row headers stick to the left side of the scroll pane at all times, regardless of how far across the matrix table the user has scrolled.

Since the JTable class doesn't have any method to only repaint a specific row or section of the table, I designed and implemented my own repaintRow method. It is used whenever the user edits a probability, as we need to update the values in the table to reflect the changes. Using the repaint method would cause the program to fetch all data for all cells in the table, which is a large amount of calculation for the user making a minor change in the table. Instead, I remove the row that is being edited, get the value of the chain from the header table, then re-use the getNoteRow method in order to get the updated version of the row and insert that new row into the position in the table where it used to be.

Since we need to find out which chain is being edited, we need the getNoteSequence method to access the header table so we know which chain and which event is being edited by the user and which chain/event needs to be updated in the matrix.

Another difficulty I faced during the implementation of the MatrixPanel was the conditional colouring of cells that reached a certain value of probability, so the higher the probability the closer to green the cell should be, while a lower probability should be closer to red. I wanted the colour change to be gradual, so if the probability was 0.4, then the colour should be 40% of the way to green. I could not get the calculations of how to transition from red to green without causing a different colour, so instead I settled with having 5 different stages of probability, being equal to 0, between 0 and 0.25, between 0.25 and 0.5, between 0.5 and 0.75, then more than 0.75, with the higher probabilities being closer to green. The conditional changing of a cells colour was done using a custom CellRenderer which checked the value it was being given and coloured the labels background of the cell to the correct value.

During the rigorous bug testing phase of the implementation stage, I discovered that users could easily enter letters or symbols into the matrix and cause the program to crash due to trying to parse a string as a double. To fix this, I had to create (yet another) custom Java Swing class, ValidatedTableModel. This class is an extension of DefaultTableModel with an overridden setValueAt

method. This overridden method uses a try catch statement that looks for a NumberFormException, which would occur if Double.parseDouble was given a string that does not fit as a double, e.g. letters or symbols. If this exception occurs, an error dialog is thrown and the user is warned. We also need to check to make sure the probability being edited isn't higher than 1, as a whole row has to equal 1. If the number is higher than 1, then we can simply treat it as if the user entered 1 and update the matrix like that.

A very late addition to this class was the functionality to zoom in/out of the matrix. I discovered this oversight in portability when I attempted to use the program on another system that has a display of 1/3rd the size of my development display. The matrix was too small to be able to see the numbers and was difficult to traverse or get a general view of. Unfortunately, due to the "incompleteness" of Java Swing, there isn't zoom functionality for JScrollPane, so I instead developed the resizeTable method which changes the row height and column width of the table to a number multiplied by a "zoom factor" that is increased/decreased by clicking on the zoom in/out buttons. Getting this to the correct amount of zoom was difficult, but seems to work well now. In addition, the maximum zoomed out table (Which is the default zoom factor) allows the user to have a nice overview of most of the matrix for analysis. It is clear to see that the pitch that follows a chain is usually close to the value of the second pitch of that chain.

With both the matrix panel and import panel finished, I moved to looking at what I could add to user customization for the score. I had originally planned to have the length of the song, the first two notes and the randomness of the score. I realized that users may want to generate a score of pitches and durations, but then change the tempo or the instrument while keeping the same song. There wouldn't really be enough space on the Import panel for so many UI components, so instead I created a new class called ScorePanel, where the users can edit parameters used to generate a score.

ScorePanel

The ScorePanel is half of the UIMain class in my old design and half designed during implementation. The first big change I made to how the user affects the score generated is added the instrument combo box. This is a dropdown menu filled with every instrument that is available to use in jMusic. It contains a focus listener that changes the instrument of the score when it loses focus. If the user generates a score, they are able to change the instrument while keeping the same song. Constructing the contents of this combo box was very difficult, as each jMusic instrument is a constant between 0 and 116, but there are gaps, such as 2, that cause a runtime error in jMusic if used. That prevented me from just having combo box index 1 = jMusic constant 1 and so on. Instead, I had to manually fill an array with each of the 116 instruments which included manually editing 116 individual strings to a form of "instrumentList[0] = PIANO". For re-use purposes, I made a new class, instrument, that contains the integer value of that constant and the name of the instrument, then used the instrument name when filling the dropdown list.

Adding the functionality to change the tempo of a score was fairly easy. Similar to the instrument, the tempo can be changed for a score without having to generate a new score.

Developing the "first notes" combo boxes also proved easier than expected. They both update the values for FIRST_NOTE and SECOND_NOTE in MainFrame, which are used whenever the user requests a new score to be generated or if a new file is imported. Due to the amount of combinations available though, I wanted some way of being able to tell if the chain has actually been

found in the imported files before. If the chain hasn't, then the music will likely slowly rise in pitch until it finds an existing chain. I decided to add a label that tells the user how many times the selected chain was found in imported files. The label is updateTODO

d by accessing the getNumOfNotesForChain method inside MainFrame.

The composition string is a great way of the user being able to edit the overall structure of the song. I have explained it in the MainFrame sub-section of this document.

I also added a new randomize button, which randomizes all parameters found on this panel within a certain range. The first notes of the song are randomly generated by picking a random note, then randomly picking another note and seeing if there's a chain until a suitable chain has been found. The tempo generates randomly between 50 and 200. The instrument combo box generates a random number between 0 and the number of instruments available and uses that. The composition string first generates a random number between 4 and 13, for the length of the string, then generates N random letters where N is the random number we previously generated.

Testing of functionality throughout the implementation stage can be found in the Test Cases document in the attached ZIP file, or in the appendix of this document labelled "Test Cases".

Evaluation

Evaluation of the end product

In order to evaluate the end product, I am using a table containing several different generations of scores and rating them on criteria specified in the evaluation design. These criteria are as follows, rated on a scale of 1-10:

- Criteria 1 is “Is the same note repeated several times? (Higher rating means less occurrences)”
- Criteria 2 is “Did the score generate any noticeable patterns from the imported files? (Higher score is more occurrences)”
- Criteria 3 is “Are notes closely grouped enough? Are there any big gaps between notes that make the section sound out of place? (Higher score is a more “normal” rhythm)”
- Criteria 4 is “Does the score maintain a relatively similar tempo or does it switch the tempo too much/too quickly? (Higher score is better flow)”
- Criteria 5 is “Does the score sound good? (Higher score is better rating)”

All of the produced scores can be found in the attached ZIP file inside “Resulting Scores/Evaluation”.

Table1: Importing only one MIDI file (Fur Elise):

Tempo = 70

First notes are the same as Fur Elise first notes (76, 75)

Instrument = Piano

Composition String = “ABCADE”

	Criteria 1	Criteria 2	Criteria 3	Criteria 4	Criteria 5	Overall
Tab1_1	10	10	8	6	8	8.4
Tab1_2	7	9	7	8	6	7.4
Tab1_3	9	10	8	8	7	8.4
Tab1_4	4	10	6	7	7	6.8
Tab1_5	8	7	10	9	3	7.4
Tab1_6	9	9	8	4	7	7.4
Tab1_7	3	7	8	6	6	6
Tab1_8	4	7	4	4	5	4.8
Tab1_9	1	10	3	3	2	3.8
Tab1_10	4	8	5	6	4	5.4
Overall	5.9	8.7	6.7	6.1	5.5	6.61

As expected, the sound produced is very predictable and often contains sections of music that are almost identical to the original score. The values for criteria 2 are very high due to this.

Table2: Importing two MIDI files (Fur Elise and Bach's Minuet):

Tempo = 90

First notes are the same as Fur Elise first notes (76, 75)

Instrument = Piano

Composition String = "ABCDEFG"

	Criteria 1	Criteria 2	Criteria 3	Criteria 4	Criteria 5	Overall
Tab2_1	6	8	8	9	6	7.4
Tab2_2	6	7	7	8	7	7
Tab2_3	5	5	5	6	4	5
Tab2_4	7	4	6	5	4	5.2
Tab2_5	4	9	7	4	5	5.8
Tab2_6	7	6	6	5	6	6
Tab2_7	9	5	4	3	4	5
Tab2_8	7	7	5	6	5	6
Tab2_9	3	6	5	5	3	4.4
Tab2_10	5	4	7	6	5	5.4
Overall	5.9	6.1	6	5.7	4.9	5.72

Adding another MIDI file to the mix severely affects the chances of producing parts similar to the original score. It also affects the quality of the song overall, as it becomes generally more random.

Table3: Importing 19 MIDI files (All are found in the attached ZIP file under “Imports”):

Tempo = 90

First notes are (60, 60)

Instrument = Piano

Composition String = “AABACD”

	Criteria 1	Criteria 2	Criteria 3	Criteria 4	Criteria 5	Overall
Tab3_1	4	5	4	4	3	4
Tab3_2	9	4	4	5	3	5
Tab3_3	6	3	2	1	4	3.2
Tab3_4	3	2	4	3	3	3
Tab3_5	6	6	5	6	5	5.6
Tab3_6	5	5	5	4	3	4.4
Tab3_7	3	4	3	3	3	3.2
Tab3_8	1	1	5	4	3	2.8
Tab3_9	5	4	3	4	4	4
Tab3_10	6	3	4	3	2	3.6
Overall	4.8	3.7	3.9	3.7	3.3	3.88

As you can see, with all files imported, the score becomes significantly worse across the board. Compared with a single MIDI file (Table1), where the rating of recognisable parts was 8.7, using all files only had a rating of 3.7. The aural quality of the song dropped significantly too, from 5.5 to 3.3.

Table4: Importing 19 MIDI files (All are found in the attached ZIP file under “Imports”):

All parameters randomized

	Criteria 1	Criteria 2	Criteria 3	Criteria 4	Criteria 5	Overall
Tab4_1	3	1	5	5	3	3.4
Tab4_2	6	6	7	6	3	5.6
Tab4_3	5	2	4	3	2	3.2
Tab4_4	6	4	4	5	4	4.6
Tab4_5	2	2	2	3	2	2.2
Tab4_6	7	2	2	5	3	3.8
Tab4_7	4	3	5	6	4	4.4
Tab4_8	3	2	4	3	1	2.6
Tab4_9	2	1	7	6	6	4.4
Tab4_10	1	4	3	3	1	2.4
Overall	3.9	2.7	4.3	4.5	2.9	3.66

As expected, the randomised cases are overall the worst performing. The aural quality of the song is only 2.9, which is likely heavily affected by the random tempo and instrument choice. The rating for repeated elements was also lower, but I suspect this is related to the use of different instruments, making it harder to notice if elements were similar to those in imported files.

Overall, I would struggle call the project a success in terms of the music generated. The system does exactly what it is intended to, but I fear that the core concept of generating music using a stochastic matrix is flawed without the addition of more processing before the score is produced, such as accompanying sounds, using left and right hands etc. The system does handle small numbers of MIDI files well but tends to fall short when using large amount of data.

As mentioned in the evaluation design, the functional requirements can be easily tested and evaluated. These are evaluated by testing in the Test Cases file in the attached ZIP file as “Test Cases”, or in the appendix of this document.

Evaluation of the project as a whole

As mentioned in the section above, I do not believe the music produced by this system is good enough to be considered successful, but I would say the project overall has been a success. The issue with the production of music is a core concept of the program. Outside of that issue, the system has been built very close to design with a relatively small number of issues and the project ran on track for the most part. In order for me to judge how I can improve on my next project undertaking, I have compiled a list of the projects strengths and weaknesses as well as problems I faced throughout the project:

- Documentation in the way of GUI elements was relatively incomplete

The design didn't contain a detailed explanation of HOW the GUI was going to work. It only had a list of classes and variables that were going to be used. This can be improved next time by writing pseudocode of exactly WHAT will happen when I click a button, or select a cell etc.

- Difficult to quantify the results of music generation

Due to the subjective nature of music, it is near impossible to quantify how well the system is working. With more time I could have done mass surveys with a rating system of how good each iteration sounds. I could also have tried to generate pieces then identify musical constructs that appear in them, such as the rule of fifths or scales.

- Cannot process masses of information (>300 MIDI files)

JMusic has a technical limitation (Or might be a bug, I'm not sure) where whenever a file is read in, it keeps the score in memory even if you're not actually storing it in your Java program. This means using ~300 MIDI files puts you at around 4gb of memory usage. In addition to this, if I were to add ~10,000 MIDI files to the matrix, the matrix would become too large to handle. An alternative to fix this would to be to use another form of data source to store details about the matrix in, such as a database or a CSV file. This would cause the program to run slower, as it would be accessing the hard drive and not the RAM, but since processing is less of a speed if the program is being used for heavy load processing it is less of a problem.

- Failed to utilise any Git software

I believed that using GitHub to manage my project would work well, allowing me to work from different places in tandem as well as being able to revert carefully to other editions of the project, but I had large issues when using it with Eclipse due to issues with the head. I tried several times to get it working, including literally copying the text content of each file into a brand new project and linking that. Eventually, I gave up on the idea of it and kept my code localised, which was inconvenient when trying to work away from home.

- Code was kept structured and commented throughout the implementation stage

This was much more useful than I originally expected. I don't usually keep code maintained in other projects, but I decided to for this project due to the length it span over. Being able to go back to the matrix code, which was built a good 2 months before I began work on the GUI, was an immense help for understanding how my system worked again.

- Time/workload was managed well

Despite workload being quite high due to being my final year of university, meaning I had to be time efficient in my tasks, I handled it well. All tasks were completed before their deadlines and to an acceptable standard.

- Program works efficiently on lower systems

Having tested the program on less powerful systems, the program still works fairly well on these lesser systems. It worked fine on the Raspberry Pi3, which only has a gigabyte of RAM and a 1.2GHz processor. This does reduce the maximum amount of files the user can import however, due to previously mention jMusic flaws.

Learning Points

Participating in this project has significantly boosted my knowledge in several key areas. It has promoted a deeper understanding of some key musical constructs as well as more experience in dealing with library. In summary, I have learnt about the following:

- **How to create music in jMusic, as well as manipulating sounds.**

Since the musical aspects of the project would be handled by jMusic, which is a Java library containing methods for creating and manipulating music, I spend a large amount of time learning how the library works and how to correctly implement it into my project. I learnt how to process a full score taken from a MIDI file and translate that into a stochastic matrix, how to generate new notes when given a set of probabilities and how to change different settings of the score such as instrument, length, or loudness. This was also my first experience using a library in a Java project, so I now know how to add libraries to projects and reference them correctly.

- **How to effectively debug and use IDEs effectively**

Debugging during implementation was a large part of the project due to the large amount of times certain methods are repeated. For instance, adding to the matrix calls the same method around 12,000 times when I have all MIDI files have been added. Most of the time I had to check the internal state of the program to ensure the matrix was working correctly. I began to use conditional breakpoints to pin down which iteration of a method call was causing an incorrect input to the matrix as well as using print statements to catch which iteration of the loop was causing exceptions such as array out of bounds in order to narrow down what the problem may be. Since the design (especially for the matrix) was near perfect, adding some methods was fairly quick as I started to use method generators in Eclipse, such as for getters and setters.

- **Ethical and copyright issues relating to intellectual property**

Previous to this project, anywhere I had used external code or copyrighted content had been a private project for experimented – not intended for public consumption, meaning I didn't really have to ask for permissions as such. In this project, however, which is intended to simulate a professional environment, I had to pay attention to what I could and couldn't use. In several places I used extracts of code from a third party author which were generally small classes intended to change behaviour of Java Swing components. In those places, I have quoted the source I took them from and checked the site for a license. In these cases, they were free to use for anyone. As for MIDI files, I found a website that had royalty free recordings of public domain compositions, allowing me to use them in any non-profit project I wanted to.

- **What a software project design document is composed of**

Although I had experience in the overall structure of project design documentation from the group project in second year, I had not produced a full collection of the documentation that was needed due to splitting the work. This time around, I created versions of every type of diagram, discussion and write up I deemed necessary for my project. This knowledge of how projects are structured and how to go about starting the design process will definitely aid me in a graduate job once I have finished my education.

- **Importance of well structure and commented code**

Since the majority of my previous projects, both education related and personal, were relatively small and implemented over a small amount of time, I never used to comment my code or concentrate too much on the structure of it until after the program was finished, where (if it was education related) I would comment the code and refactor it. I decided to try and keep to code commented and well-structured throughout the implementation stage as I would be working on it over the course of several months. This worked very well, especially in the case of the stochastic matrix. I worked on the stochastic matrix then move onto doing other work before coming back to the project a whole month later. The complexity of the matrix, which spans over a hierarchy of 6 classes, would have been near impossible to figure out again if it wasn't for the comments.

Professional Issues

As the project was undertaken in a similar to manner to a project run by a business, I aimed to meet the standards set by the BCS (British Computing Society) as to avoid any ethical, moral or professional issues during the duration of the project. Listed here are some examples of the codes and conducts that were followed/enforced, as well as examples of how they were met:

- **"Wherever possible, avoid platform-specific techniques that will limit the opportunities for subsequent upgrades."** 5.2 Software Development: When Programming

Since the program works through Java, which runs in a virtual machine environment, the code is automatically fairly portable. It runs on Linux and Mac, since UI elements are controlled by the operating systems default look and feel.

- **"Encourage re-usability; consider the broader applications of your designs and, likewise, before designing from new seek out any existing designs that could be re-used."** 5.2 Software Development: When Designing Software

Since the possibility of using the stochastic matrix in other applications is relatively high, I took steps to make sure that the matrix is completely disjoint from the rest of the program and could be ported to other systems fairly easily. This also applies to the visualisation of the matrix, which is a JPanel that can simply be added as a new class and can be grab data from a matrix or another data source set up correctly.

- **"Strive to produce well-structured code that facilitates testing and maintenance."** 5.2 Software Development: When Programming

I ensured that I followed this step throughout the implementation stage as it is much easier to build-on from, debug and edit existing code if it is properly indented, commented and refactored. Specific examples include a method that sets up all action listeners, rather than including them in the constructor of a panel class to reduce code-clutter, and the method in MainFrame that generates a grid bag constraints class for building with buttons (which also reduces code duplication). Keeping early code commented (in this case the matrix) helped significantly, as I was easily able to get back to the old classes to check if they were working as intended and fix them if needs be.

- **"Avoid unnecessary work by researching previous problems and looking for common solutions."** 5.2 System Installation: When Scheduling Installation Work

Unfortunately, due to the popularity of the jMusic package, I struggled to find much troubleshooting information online. I did, however, find a large resource of questions, problems and solutions to Java Swing issues. I accessed these frequently to try and troubleshoot my program, especially when building the JTable filled with the data from the matrix, issues such as validated the table entries, adding a header to each row and only allowing the selection of one row. These are all common problems in Java Swing which many people have already come up with inventive solutions/bug fixes, saving me a large amount of time and effort as well as making the program more efficient.

- **“Do not rely solely on the direct outputs of tests, but check values are as expected in internals tables, databases and error logs.”** 5.2 Software Development: When Testing

While testing the matrix during implementation it was not really possible to test through output to the screen (such as print statements) since the system needs to have a correct internal state. In order to verify the values for this internal state (specifically the matrix) I had to use breakpoints and manually search through the hierarchy of arrays that represent the matrix. By doing this, I could verify if the matrix had the correct values where they should have been updated.

Bibliography

[1] p71, Nierhaus, G., 2009. Algorithmic composition: paradigms of automated music generation. Springer, Wien ; New York.
[2] p364, Dodge, C., Jerse, T.A., 1997. Computer music: synthesis, composition, and performance, 2. ed. ed. Schirmer Books [u.a.], New York.
[3] Xenakis, I., n.d. Stochastic Composition and Stochastic Timbre: GENDY3.
[4] Brown, Brown Andrew R. "jMusic Tutorials," n.d. http://explodingart.com/jmusic/.

Brown, A.R., 2005. Making music with Java: an introduction to computer music, Java programming and the jMusic library. Brisbane.
de la Higuera, C., Fédéric, P., Frédéric, T., n.d. Learning Stochastic Finite Automata for Musical Style Recognition.
Fritsch, E.F., Viccari, R.M., n.d. CAMM - Automatic composer of musical melodies.
Marcus T., P., Geraint A., W., n.d. Improved Methods for Statistical Modelling of Monophonic Music.
Papadopoulos, G., Wiggins, G., n.d. AI Methods for Algorithmic Composition: A Survey, a Critical View and Future Prospects.
Pedro, P.C., Enrique, V., n.d. Learning Regular Grammars to Model Musical Style: Comparing Different Coding Schemes. (1998).
Roads, C., 1996. The computer music tutorial. MIT Press, Cambridge, Mass.
Tavares, T.F., n.d. Computational Models for Rhythm and Applications on Human-Machine Interactions (2013).
Verwer, S., Eyraud, R., de la Higuera, C., n.d. PAutomaC: a probabilistic automata and hidden Markov models learning competition.

Appendix

Test Cases:

Test Type	Section	Test Information	Expected Result	Actual Result	Comments
Functional Testing	Import Panel	Able to import a MIDI file	File appears in table and matrix is updated accurately	Working	
		Able to import multiple MIDI files	Files appear in table and matrix is updated accurately	Working	
		Importing file that has already been importing	Throw an error	Not Working	Was not throwing an error, does nothing. Since updated to throw an error.
		Removing a single imported MIDI file	Remove the selected file and update the matrix	Working	
		Removing multiple imported MIDI files	Remove the selected files and update the matrix	Working	
		Importing a non MIDI file	Throw an error telling the user the file is not a MIDI file	Working	
		Click view matrix button	Show the matrix view panel	Working	
		Click Info button	Show info panel	Working	
		Click generate button	Show score manager panel	Working	
		Remove all imports then try to remove again	Remove button should become disabled	Working	
		Import 300 MIDI files	Import files and update matrix	Not Working	I looked into why the program would crash with large amounts of MIDI files, and it turns out that issues with memory reserves in jMusic prevent large amount of songs from being read in. Not fixible.
		Click generate with no files imported	Throw an error dialog	Working	
		Click view matrix with no files imported	Throw an error dialog	Working	
		Click close button	All process should be exited	Working	Checked for remaining processes in Windows task manager
	Matrix Panel	Click back button	Show import panel	Working	
		Zoom in to max	Show matrix at largest form	Working	Still readable at 1280x800 resolution
		Zoom out to max	Show matrix at smallest form	Working	Not readable (Intended to just show overview) at 1280x800 resolution, but can still see structure
		Increase a value from 0 to 0.25 when another cell in the same row contains 1	Probability at 1 should go to 0.75, other probability should go to 0.25	Working	
		Decrease a value from 0.25 to 0 when another cell in the same row contains .75	Probability at .75 should go to 1, other probability should go to 0	Working	
		Change a probability to 0.76	Colour should go brightest green	Working	
		Change a probability to 0.51	Colour should go slightly darker green	Working	
		Change a probability to 0.26	Colour should go very dark green/dark red	Working	
		Change a probability to 0.01	Colour should go dark red	Working	
		Change a probability to 0	Colour should go darkest red	Working	
		Change a probability to "A"	Error dialog should pop up	Working	
		Change a probability to "#"	Error dialog should pop up	Working	
		Change a probability to 1.01	Probability should not change (Rows cannot be higher than 1)	Working	
	Score Manager Panel	Click generate score button	Score displayed by generate score should be different than previous	Working	Tested multiple times
		Click randomize score	All parameters on panel should randomize, first notes should be a successful chain	Working	Tested 30 times to ensure
		Set tempo to 49	Throw an error dialog since accepted values are 50-200	Working	
		Set tempo to 201	Throw an error dialog since accepted values are 50-200	Working	
		Change score to a different instrument	Score should play in a different instrument	Working	Tested all 116 instruments, no run time errors and each is unique
		Enter 200 character string in composition field	Score generated should be very long	Working	
		Change first notes to 66, 80	First two notes in score should be 66, 80	Working	
		Import fur elise and change first notes to 76, 75	Total count should equal 15	Working	
		Change composition string to "AAAAAAAAAB" and click generate	All phrases in score should be the same except for the last phrase and the first two notes	Working	
		Click view score	jMusic score window should display with the score in it	Working	
		Change composition string to "@@" and click generate	Error dialog should pop up since symbols are not allowed	Working	
		Change composition string to "á"	Error dialog should pop up since non english alphabet letters are allowed	Working	
		Change composition string to "1"	Error dialog should pop up since numbers are not allowed	Working	
		Save score to MIDI file	MIDI file should be created in directory pointed to and should contain the score we tried to save	Working	