

Shawn Huntzberry

Capstone Design I

Component Design HW 2

8 December 2015

Artificial Instructor

Pesudocode for Components:

- Audio Input/Output:
 - Variables:
 - Mixer mix
 - int sampleRate
 - int bufferSize
 - int overlap
 - AudioFormat format
 - DataLine.Info info
 - TargetDataLine outLine
 - SourceDataLine inLine
 - AudioInputStream stream
 - byte [] data
 - InputStream byteArrayIS
 - ByteArrayOutputStream out

sampleRate = 4096;

bufferSize = 1024;

overlap = 0;

format = new AudioFormat(sampleRate, 16, 2, true, true);

mix = AudioSystem.getMixer("Default");

//set up for input

info = new DataLine.Info(TargetDataLine.class, format);

line = (TargetDataLine) mix.getLine(info);

//open the line for input

outLine.open(format, sampleRate);

outLine.start();

stream = new AudioInputStream(outLine);

```

//reads in data
outLine.read( data, 0, 1024);
out.write( data, 0, 1024)

//set up output of data and open line
info = new DataLine.Info( SourceDataLine.class, format );
inLine = ( SourceDataLine ) AudioSystem.getLine( info );
inLine.open( format, 1024 );

//start line and write data to it
inLine.start();
inLine.write( data, 0, 1024 );
inLine.drain();
inLine.close();

```

- Pitch Detection & Note Assignment
 - Variables:
 - int octave
 - String noteValue, note
 - float pitch
 - JVMAudioInputStream audioStream
 - AudioDispatcher dispatcher
 - PitchEstimationAlgorithm algo

Pseudocode starts after audio input has been set up to the point of AudioInputStream

```

algo = PitchEstimationAlgorithm.FFT_YIN;
audioStream = new JVMAudioInputStream(AudioInputStream);
dispatcher = new AudioDispatcher(audioStream, BUFFER_SIZE, 0);
dispatcher.addAudioProcessor(new PitchProcessor(algo, SAMPLE_RATE, BUFFERSIZE, this);
new Thread(dispatcher, "Audio Dispatching").start();

```

```

//Starts running the method handlePitch(PitchDetectionResult pitchDetection, AudioEvent event)
handlePitch( PitchDetectionResult pitchDetection, AudioEvent event )
{
    if( pitchDetection.getPitch() != -1 )

```

```

{
    pitch = pitchDetection.getPitch();
    String noteValue = getNoteValue(pitch, 0);          //jumps to getNoteValue method
    PRINT OUT pitch and noteValue
    UPDATE FRETBOARD with note value marked
}
}

```

String getNoteValue(float pitchValue, float tuningOffset)

```

{
    String retVal;
    //set of conditionals to get the note value based on passed in pitch
    if( 0 < pitch <= X )
    {
        //set global variable octave to corresponding octave
        octave = 0;
        if ( 0 < pitch <= X1 )
        {
            retVal = CORRESPONDING_NOTE_VALUE;
        }
        else if ( X1 < pitch <= X2)
        {
            retVal = CORRESPONDING_NOTE_VALUE;
        }
        else if(.....){ ..... }
    }
    else if( X < pitch <= Y )
    {
        //set global variable octave to corresponding octave
        octave = 1;
        if ( 0 < pitch <= Y1 )
        {
            retVal = CORRESPONDING_NOTE_VALUE;
        }
        else if ( X1 < pitch <= Y2)
        {

```

```

        retVal = CORRESPONDING_NOTE_VALUE;
    }
    else if(.....){ ..... }
}
else if( Y < pitch <= Z )
{
    //set global variable octave to corresponding octave
    octave = 2;
    if ( 0 < pitch <= Z1 )
    {
        retVal = CORRESPONDING_NOTE_VALUE;
    }
    else if ( X1 < pitch <= Z2)
    {
        retVal = CORRESPONDING_NOTE_VALUE;
    }
    else if(.....){ ..... }
}
....

return retVal;
}

```

- Visualization

- Oscilloscope

- Variables:

- JVMAudioInputStream audioStream
 - AudioDispatcher dispatcher
 - OscilloscopePanel oPanel

Pseudocode starts after audio input has been set up to the point of AudioInputStream

```

audioStream = new JVMAudioInputStream(AudioInputStream);
dispatcher = new AudioDispatcher(audioStream, BUFFER_SIZE, 0);
dispatcher.addAudioProcessor(new Oscilloscope(this));
new Thread(dispatcher, "Audio Dispatching").start();
OscilloscopePanel oPanel = new OscilloscopePanel();

```

```
//Jumps to handleEvent(float [] data, AudioEvent event)
```

```
void handleEvent( float [] data, AudioEvent event)
```

```
{
    for( i = 0 ; i < data.length ; i++ )
    {
        data [ i ] = data [ i ] * 5 ;
    }
    oPanel.paint( data, event );
    oPanel.repaint();
}
```

- Fretboard Visualization

- Variables

- FretBoardPanel fretBoardPlayer
 - String noteVal
 - int octave
 - Graphics g
 - boolean allNotes

Pesudocode starts at the point after the note value has been stored from the pitch value

```
fretBoardPlayer = newFretBoardPanel();
```

```
g = fretBoardPlayer.getGraphics();
```

```
//jumps to the handlePitch method to constantly update
```

```
void handlePitch( PitchDetectionResult pitchDetection, AudioEvent event )
```

```
{
    noteVal = getNoteValue( pitchDetection.getPitch(), 0 );
    fretBoardPlayer.setNoteDisplayMode( allNotes ); //if allNotes false shows single note,
                                                    //else it shows all possible notes

    fretBoardPlayer.setNoteVal( noteVal );
    fretBoardPlayer.setOctave( octave );
    fretBoardPlayer.repaint(); //jumps to repaint method in class
}
```

```
void paintNotes( Graphics g )
```

```
{
    if ( allNotes == true )
```

```

{
    if ( noteVal == "A" )
    {
        //paint all possible notes for A
        g.fillOval(X, Y, R, R);
        g.drawString("A", X1, Y1);
    }
    else if( noteVal == "B" )
    {
        //paint all possible notes for B
        g.fillOval(X, Y, R, R);
        g.drawString("B", X1, Y1);
    }
    else if( ... ) { ... }
}
else
{
    if ( noteVal == "A" )
    {
        //paint all possible notes for A
        g.fillOval(X, Y, R, R);
        g.drawString("A", X1, Y1);
    }
    else if( noteVal == "B" )
    {
        //paint all possible notes for B
        g.fillOval(X, Y, R, R);
        g.drawString("B", X1, Y1);
    }
    else if( ... ) { ... }
}
}

```

- Feedback

Pesudocode starts after input has been set up

```
note = getNoteValue( pitchDetection.getPitch(), 0 );  
time = event.getTimeStamp();
```

```
//start by comparing time in lesson to time the note value was strummed
```

```
if( 0 < time <= expectedTime[ X ] )
```

```
{
```

```
    //now check the note value
```

```
    if( note == expectedNote[ X] )
```

```
    {
```

```
        //increase the user's score
```

```
        userScore++;
```

```
    }
```

```
    else
```

```
    {
```

```
        //decrease the user's score
```

```
        userScore--;
```

```
    }
```

```
}
```

```
else if ( expectedTime[ X ] < time <= expectedTime[ Y ] )
```

```
{
```

```
    //now check the note value
```

```
    if( note == expectedNote[ Y] )
```

```
    {
```

```
        //increase the user's score
```

```
        userScore++;
```

```
    }
```

```
    else
```

```
    {
```

```
        //decrease the user's score
```

```
        userScore--;
```

```
    }
```

```
}
```

```
else if( ... ){ ... }
```

Major Algorithms of my Program:

- Strumming algorithm

Looking at audio data input, into a `ByteArrayOutputStream`, we convert it to a usable byte array. Using a for loop the algorithm find the major peaks, both positive and negative, in order to determine when a note is simply ringing out and when a new note has been struck. By looking at the different peaks the algorithm is able to determine differences in the amplitude that would signify a new note. If the peak remains at 0 we know that no note has been strummed, and if a peak is below a certain threshold then it is not a note that has been strummed but rather excess noise that was picked up. The problem with this implementation is the fact that we are looking at data in different sample sizes so if a note rings at the end of one sample size and continues into the next sample size the peak will be represented in both, when in fact it is the same note. In order to solve this I will utilize an Linked List data structure in order to be able to store a fixed amount of sample sizes to be able to look back at the last sample size and compare some peaks.

- Note Assignment

This algorithm relies heavily on the use of the `PitchDetectionHandler` and the result for the `handlePitch` method it runs. `handlePitch` yields the pitch value calculated and from that my created method `getNoteValue` returns the octave and note value that corresponds to that pitch. In order to do this I have a series of possible pitch ranges that relate to the different note values and octaves. By a series of conditional statements I first check the overall range in order to locate the correct octave. After the octave is matched and the program steps into that conditional, it goes through another series of conditionals to find where the input pitch corresponds to. Once the appropriate range is located the note value is set and output to the screen.

- Lesson Comparison

Lesson comparison will be used by creating the data structure `LessonContent`. `LessonContent` will contain arrays that contain the information about the value of expected notes, the pitch of expected notes, and the timing of the notes. When a user attempts a lesson the time of each note struck is first compared to the time in the expected lessons array of times. After the times are matched up, between the input and expected result, interval at which that time is located in the array is stored. Using this interval the program compares the value in the expected note value array to the value of the user's input at that time. If they match then a score is assigned. Depending on accuracy, the user's score is updated.

- Scoring Algorithm

The scoring algorithm can both add and remove points from a user's score based on input. The algorithm will increase the user's score for the correct note value at the corresponding time range in the lesson. The algorithm will decrease the user's score if an incorrect note is played during a time range. This algorithm has some room for error, as the user will increase their score for just hitting the correct note as much as possible within the expected time frame. However, the algorithm compensates for this by allowing for a user's score to decrease if extra notes are played during a time period.

Data Structures(unique to my program):

- `PlayBackThread` facilitates the real-time audio output while a user is playing. The playback thread is the `Runnable` job that a thread is set to. It takes in two parameters, a `SourceDataLine` and the `AudioInputStream`. With this information it runs in order to output data as the thread is alive.
- `FretBoardPanel` is the data structure to take in data and hold the values and the graphics for visualization. The class has three global variables that are set using the corresponding methods. The variables are `displayAllNotes`, `currentOctave`, and `noteValue` and are set using `setNoteDisplayMode(boolean mode)`, `setOctave(int octave)`, and `setNoteValue(String note)`. Using this information it uses constant values to calculate and update the appropriate graphics to display to the user.
- `Lesson` is a data structure that holds the information about the different lessons. It holds the information about the lesson and the values necessary to calculate feedback and scores for the users attempt. The values it stores are the lesson number, the lesson title, the level of the lesson, the current high score, an array of the times each note occurs, and an array of the corresponding note values at those times. These values may be set and retrieved using the getter and setter methods in the `Lesson` class.

Screens of the Application:

Below: Home Screen

Welcome <USERNAME>!

<u>Lessons Completed:</u> XXXX	<u>Current Score:</u> XXXX
<u>Highest Score:</u> XXXX	<u>Last Lesson:</u> LEVEL X, LESSON X
<u>Current Level:</u> XXXX	<u>Total Time:</u> X Hours X Minutes

Enter Classroom

Use Tuner

Below: Tuner Screen

[illegible]

Below: Classroom(Levels) Screen

Welcome to the Classroom: Please Select a level to continue

Level 1	Level 2	Level 3	Level 4
Level 5	Level 6	Level 7	Level 8
Level 9	Level 10	Level 11	Level 12

Back To Home Screen

Below: Lesson Section Screen

Welcome to Level XX: Please Select a lesson to continue

XXXX points needed to unlock next level!

Lesson 1 Score: 96	Lesson 2 Score: 75	Lesson 3 Score: 99	Lesson 4 Score: 84
Lesson 5 Score: 101	Lesson 6	Lesson 7	Lesson 8
Lesson 9	Lesson 10	Lesson 11	Lesson 12

Back to Classroom

Back to Home Screen

Below: Lesson Attempt Screen

A (37.897 Hz)
G (89.862 Hz)

Start

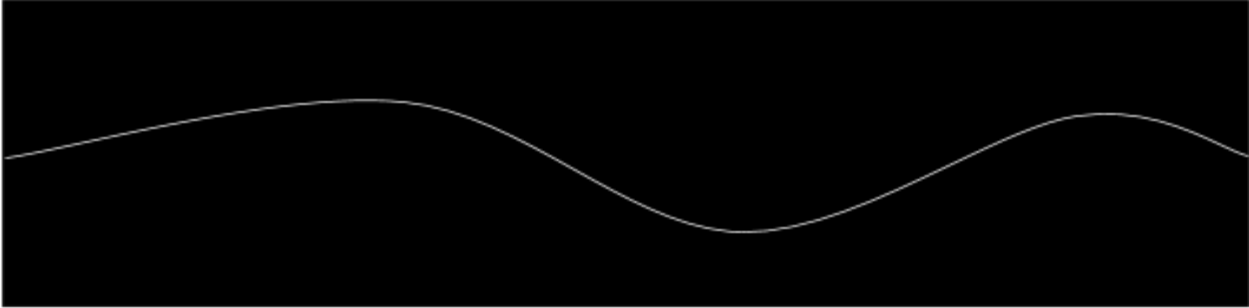
Stop

Tuner

Show Single Note

Open 1 2 3 4 5 6 7 8 9 10 11

G												
D												
A												
E												



Key: ● = expected note to be played ● = note played by user

Below: FEEDBACK FOR ATTEMPT

Feedback for level X, lesson X:

User Statistics

Score: 74

Notes Correct: 90

Notes Incorrect: 26

Previous High-Score: 71

Verdict: PASSED

Lesson Statistics

Highest Possible Score: 110

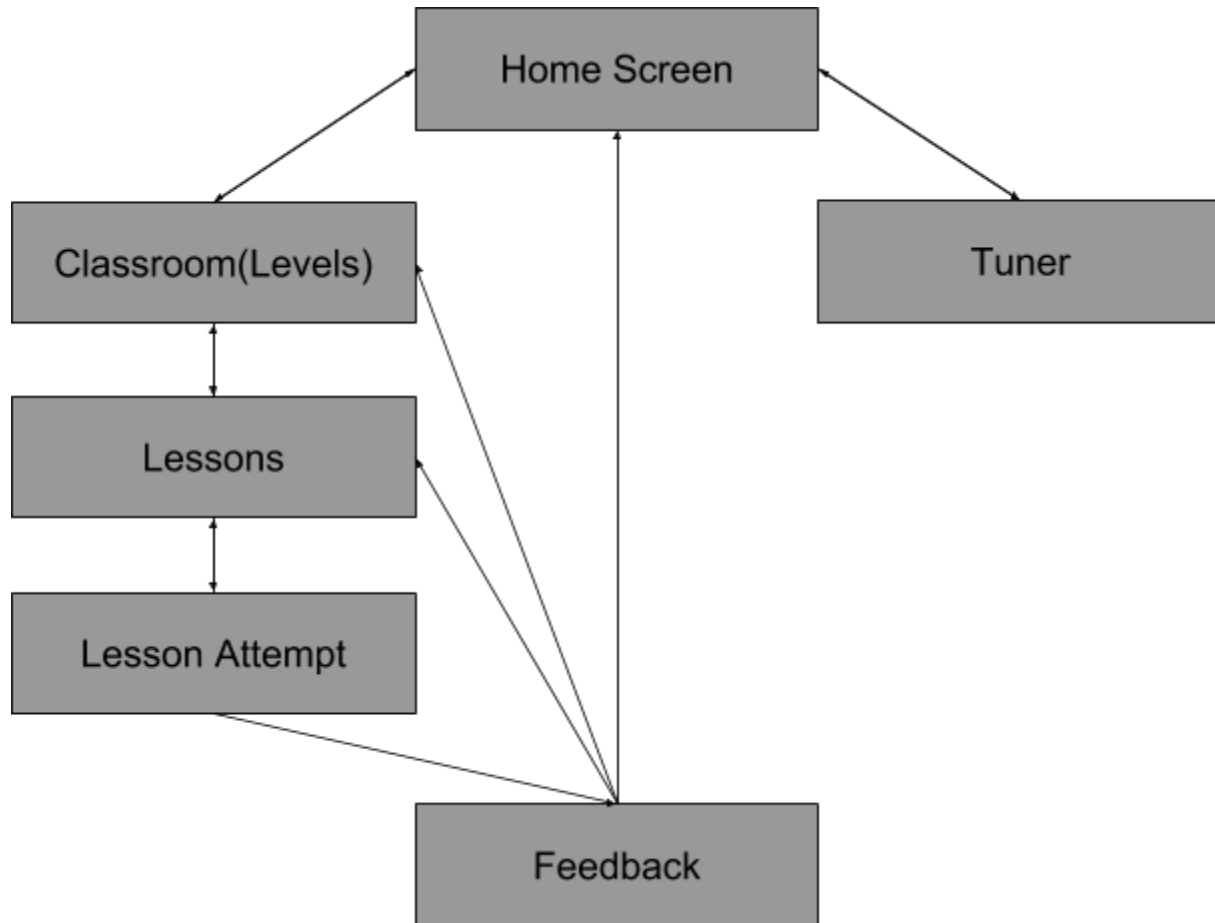
Number of Notes: 110

Lowest Passing Score: 74

Index of note - - - - Note Played - - - - Time Stamp - - - - Expected Note

1 2 3	A B D	1.36s 4.55s 9.10s	A B C
--------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------

Workflow through the screens:



External Interfaces and Data Structures:

- JavaSound API:

The JavaSound API is a major aspect of my program. I utilize the Mixer, DataLine, TargetDataLine, SourceDataLine, AudioFormat, and AudioInputStream data structures and interfaces. The Mixer data structure allows the program to scan the system for available sound inputs and outputs using the built in "getMixer()" method. I am then able to set format for reading the audio data by using the AudioFormat structure. To set the format I must specify the sample rate, sample size in bits, number of channels, if it is signed or not, and if it is big endian. Once the format is set and the mixer is chosen I can update the DataLine.Info interface in order to add media functionality to the input or output line specified. After the line is set I am able to cast the line to be either input or output by using the TargetDataLine(input) or SourceDataLine(output) interfaces. After all of that is complete I am able to set either dataline to the current AudioInputStream in order to read in and store data.

- TarsoDSP library:

From the TarsoDSP library I will be using the JVMAudioInputStream, AudioDispatcher, AudioProcessor, PitchProcessor, and Oscilloscope data structures. The JVMAudioInputStream allows the AudioInputStream data structure to be transformed into a usable input stream that the TarsoDSP methods can use. The AudioDispatcher data structure takes in an AudioProcessor and constantly runs a job. The AudioDispatcher is the thread instance and the AudioProcessor is the runnable job. The PitchProcessor and Oscilloscope are two subclasses of the AudioProcessor that are unique to different functions of the TarsoDSP library. The PitchProcessor takes in a PitchEstimationAlgorithm, sample rate, buffer size, and the PitchDetectionHandler. Once this is created, and assigned to a dispatcher, the PitchDetectionHandler will constantly run the method handlePitch in order to output the frequency of the audio input using the algorithm that was passed in. The Oscilloscope subclass is also assigned to a dispatcher but only requires an OscilloscopeEventHandler be passed in. The OscilloscopeEventHandler runs the method handleEvent which maps the data that is input, using the interface OscilloscopePanel

The only algorithm I have used is “FFT_YIN” offered as a type of PitchEstimationAlgorithm. It estimates the frequency of the audio input based on the YIN model of pitch detection and is the most effective for programs similar to mine.

The only external interface I will use in my application is the OscilloscopePanel offered by the TarsoDSP library. This panel takes the input from the JVMAudioInputStream and graphs the soundwave for the user to see. Using this I will be able to show the user the appearance of their sound input to show when they let a note ring out and when they strike a note.