Criterion C: Development

Techniques Used

Graphical User Interface (GUI)

The software I've created is entirely focused on the user interface. This results in a complex relationship between adding notes to the GUI. The **addNote()** method shown below is responsible for both the graphical input of a note, as well as adding a note to their respective LinkedList.

```
public void addNote(int duration, int y, String img, String i) {
    /*
     * FUNCTION:
     * Plays a demo of the note added.
     * Creates a new MusicNote to be graphically added AND added to a LinkedList.
     * Checks if the image of a note needs to be inverted.
     * Checks if a note fits in the current measure. If not, sends an error message.
     */
    MusicNote mn = null;
    pitchSelected = basePitch + octaveMultiply;

    if (trebleBeats + duration > 8 || bassBeats + duration > 8) {
        //Error noise.
        System.out.println("You cannot enter that note here.");
    } else {
        switch (duration) {
            case 1:
            if (checkOctave != "" && checkNote != "") {
                Note n = new Note(pitchSelected, EN);
                Play.midi(n);
                if (pitchSelected > 72 || pitchSelected > 52 && pitchSelected < 60) {
                    mn = new EighthNote(pitchSelected, setYCoordinate(pitchSelected)+33);
                    mn.setImage(i);
                } else {
                    mn = new EighthNote(pitchSelected, setYCoordinate(pitchSelected)+1);
                    mn.setImage(img);
                }
            } else {
                Note n = new Note(60, EN);
                Play.midi(n);
                mn = new EighthNote(pitchSelected, 221);
                mn.setImage(img);

            }
            break;
```

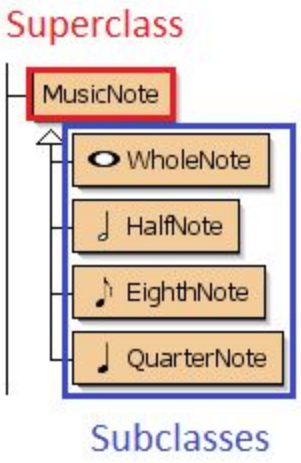
```
case 2:
if (checkOctave != "" && checkNote != "") {
    Note n = new Note(pitchSelected, EN);
    Play.midi(n);
    if (pitchSelected > 72 || pitchSelected > 52 && pitchSelected < 60) {
        mn = new QuarterNote(pitchSelected, setYCoordinate(pitchSelected)+33);
        mn.setImage(i);

    } else {
        mn = new QuarterNote(pitchSelected, setYCoordinate(pitchSelected));
        mn.setImage(img);
    }
} else {
    Note n = new Note(60, EN);
    Play.midi(n);

    mn = new QuarterNote(pitchSelected, 220);
    mn.setImage(img);
}
break;
```

```
case 4:
if (checkOctave != "" && checkNote != "") {
    Note n = new Note(pitchSelected, EN);
    Play.midi(n);
    if (pitchSelected > 72 || pitchSelected > 52 && pitchSelected < 60) {
        mn = new HalfNote(pitchSelected, setYCoordinate(pitchSelected)+33);
        mn.setImage(i);
    } else {

        mn = new HalfNote(pitchSelected, setYCoordinate(pitchSelected));
        mn.setImage(img);
    }
} else {
    Note n = new Note(60, EN);
    Play.midi(n);

    pitchSelected = 60;
    mn = new HalfNote(pitchSelected, 220);
    mn.setImage(img);
}
break;

case 8:
if (checkOctave != "" && checkNote != "") {
    Note n = new Note(pitchSelected, EN);
    Play.midi(n);

    mn = new WholeNote(pitchSelected, setYCoordinate(pitchSelected)+16);
    mn.setImage(img);
} else {
    Note n = new Note(60, EN);
    Play.midi(n);

    mn = new WholeNote(pitchSelected, 236);
    mn.setImage(img);
}
break;
```

I used multiple **conditional statements**, including ***if* statements nested within *switch* statements** in the above method. My **justification** for doing so is to allow for flexibility for the user; for example, the user will be able to add any kind of note to the graph by calculating certain coordinates using the conditional statements. The switch statements can determine the image file, pitch, rhythm, and graphical coordinates of any note. Thus, the inclusion of this complexity is useful for the functionality of the software.

Inheritance

MusicNote is an important superclass in this application. Inheritance is essential for the software because it reduces redundancies in the code for several important methods.



Essentially, the MusicNote superclass contains the necessary fields for its subclasses. This is **justified** because all MusicNote classes will have the same 7 fields, but with different values being passed into them. This reduces the need to reuse code, and thus reduces the risk of bugs from inconsistent code.

Data Structures

Because of the nature my application, and the use of the jMusic external library, I have used two main data structures: **LinkedLists** and **Vectors**.

Although jMusic is a musical external library that structures music data in a highly organized fashion, it is also considerably outdated, starting out in the late 1990s. As a result, the essential classes of jMusic, such as Phrases, Parts and Scores, can only hold Arrays or Vectors as a data structure. Furthermore, **Note** objects in jMusic can only pass parameters in pitch frequency and rhythm value. This is not entirely applicable to the nature of my application, because my **MusicNote** objects require other arguments.

1.

```
//The following Vectors will hold jMusic Note objects:
Vector trebleJM = new Vector();
Vector bassJM = new Vector();
```

Vectors for jMusic *Note* objects are initialized in the World.

2.

```
public MusicStaff()
{
    // Create a new world with 6
    super(1000, 600, 1);
    setupInventory();
    totalTrebleBars = 0;
    totalBassBars = 0;
    trebleBeats = 0;
    bassBeats = 0;
    upperStave = false;
    lowerStave = false;

    phraseA = new Phrase(0.0);
    phraseB = new Phrase(0.0);
    partA = new Part();
    partB = new Part();
    score = new Score();

    partA.addPhrase(phraseA);
    partB.addPhrase(phraseB);
    score.addPart(partA);
    score.addPart(partB);
}
```

The World's constructor organizes jMusic classes to set up an empty music file for the user to make inputs.

3.

```
public void playMelody() {
    phraseA.addNoteList(trebleJM, false);
    phraseB.addNoteList(bassJM, false);

    Play.midi(score);
}
```

When the user plays the existing melody, the vectors of Note objects are added to the jMusic *Phrase* class.

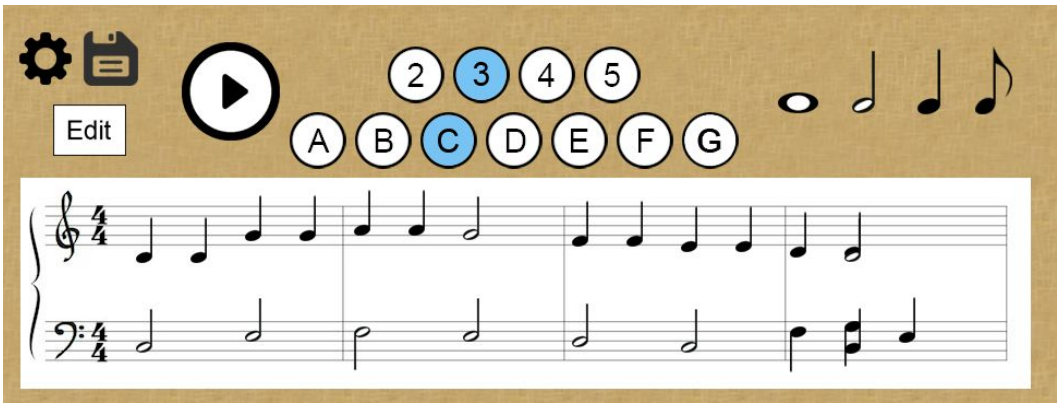
Now the notes are part of the music, and the application plays this as a MIDI sound file.

As a result, my **justification** for also using LinkedLists is that my software also requires graphical input. This is why the **MusicNote** class contains more than just a pitch and rhythm argument, but arguments for an image file and XY coordinates as well.

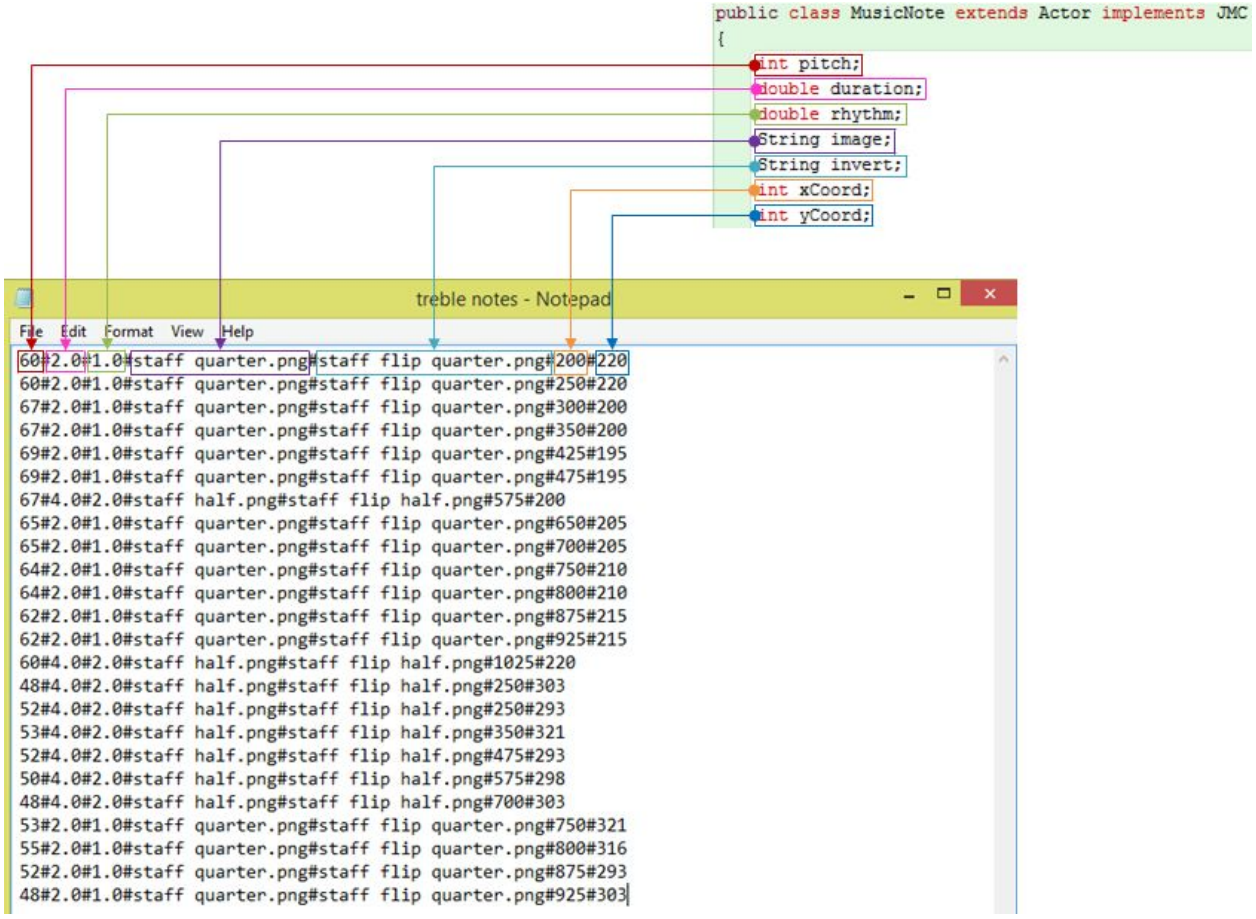
Ultimately, they are both appropriate data structures for my application, due to the dynamic ability for the data structure to grow. This is necessary because it is unknown how many notes a user will add; composing a piece of several eighth notes requires more elements to be added to a LinkedList than a piece of several whole notes.

File Input and Output

In order to save the current melody to file, MusicNote objects are saved into a #-delimited list on a text file.



Aside from a few graphical issues, the above screenshot showcases an excerpt of *Twinkle Twinkle Little Star.* Below visualizes how the melody is saved to file.



Each note displayed on the GUI is saved as a **MusicNote** object, each with a specified pitch, duration, etc. This is saved to a text file using a #-delimited list, which separates each of the object's properties. Thus, each line on the text file represents the data for a single note.

```
void write() {
    MusicNote[] tArray = trebleNotes.toArray(new MusicNote[trebleNotes.size()]);
    MusicNote[] bArray = bassNotes.toArray(new MusicNote[bassNotes.size()]);

    try {
        writer = new PrintWriter(new FileWriter(FILE_PATH));
        for (int i = 0; i < tArray.length; i++) {
            writer.println(tArray[i].getPitch() + "#" + tArray[i].getDuration() + "#" + tArray[i].getRhythm() + "#"
            + tArray[i].getImg() + "#" + tArray[i].getInvert() + "#" + tArray[i].getXCoord() + "#" + tArray[i].getYCoord());
        }

        writerB = new PrintWriter(new FileWriter(FILE_PATH_BASS));
        for (int j = 0; j < bArray.length; j++) {
            writer.println(bArray[j].getPitch() + "#" + bArray[j].getDuration() + "#" + bArray[j].getRhythm() + "#"
            + bArray[j].getImg() + "#" + bArray[j].getInvert() + "#" + bArray[j].getXCoord() + "#" + bArray[j].getYCoord());
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    writer.close();

    System.out.println("Saved successfully.");
}
```

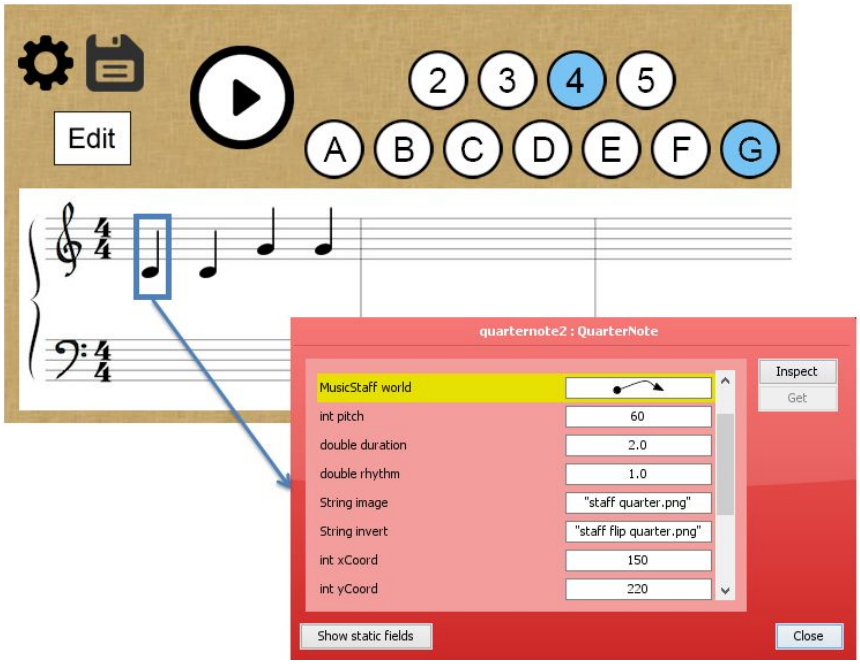
Following up on the use of data structures, the LinkedLists are converted into Arrays in the write() method in order to properly write to a text file. This makes it easier to copy note properties through a **for-loop**.

```
void read() {
    LinkedList<MusicNote> readNotes = new LinkedList();
    String nextLine;
    String[] splitLine = new String[6];
    int count = 0;

    try {
        reader = new BufferedReader(new FileReader(FILE_PATH));
        nextLine = reader.readLine();
        while (nextLine != null) {
            splitLine = nextLine.split("#");
            MusicNote mn = null;
            int pit = Integer.parseInt(splitLine[0]);
            double dur = Double.parseDouble(splitLine[1]);

            System.out.println(dur);
            int xC = Integer.parseInt(splitLine[5]);
            int yC = Integer.parseInt(splitLine[6]);
```

Finally, the **read()** method splits each line of a text file into the appropriate fields of a MusicNote, and once again uses an algorithm similar to the addNote() method to add MusicNote objects to the GUI.

Debugging

One method of debugging that I used in Greenfoot is the use of the **Greenfoot Object Inspector**. This tool allows me to check the state of an object's fields at any point when the software isn't running. For example, this has helped me solve problems with adding notes to the GUI with correct coordinates, by simply inspecting a MusicNote object's various coordinate-related fields.

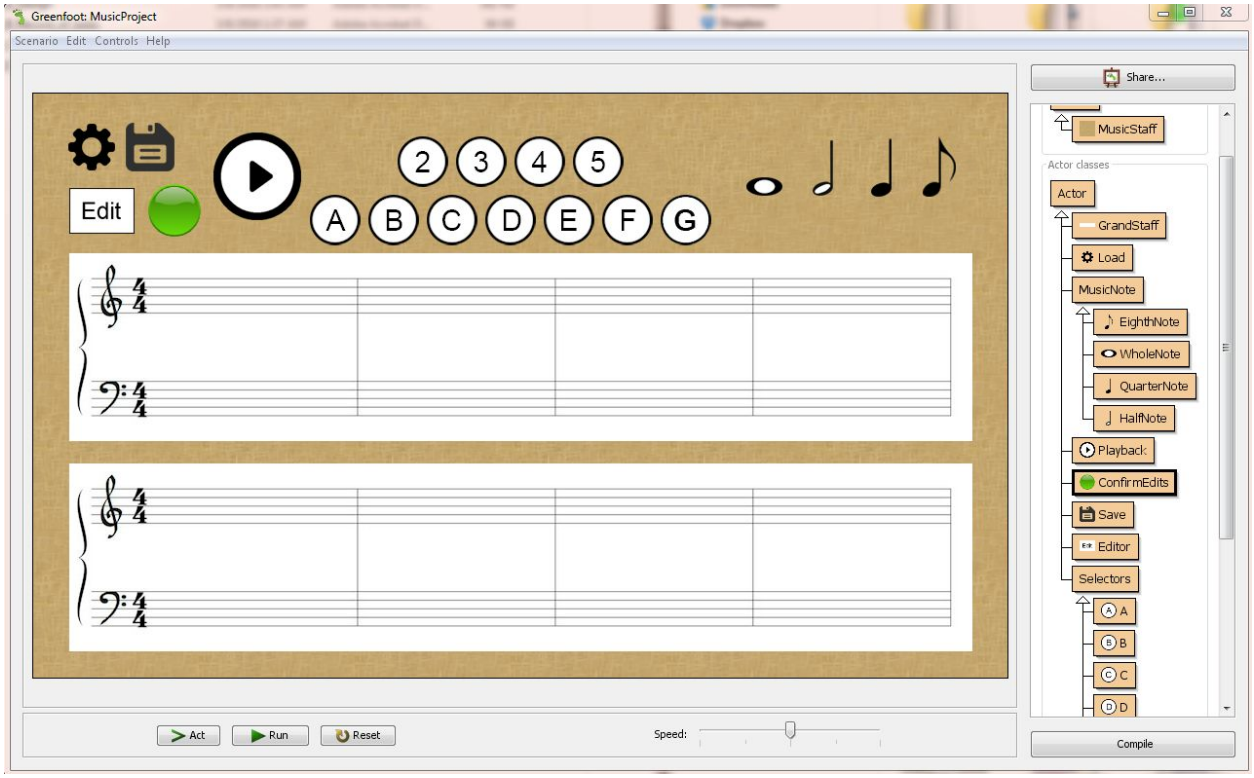


Third-Party Tools

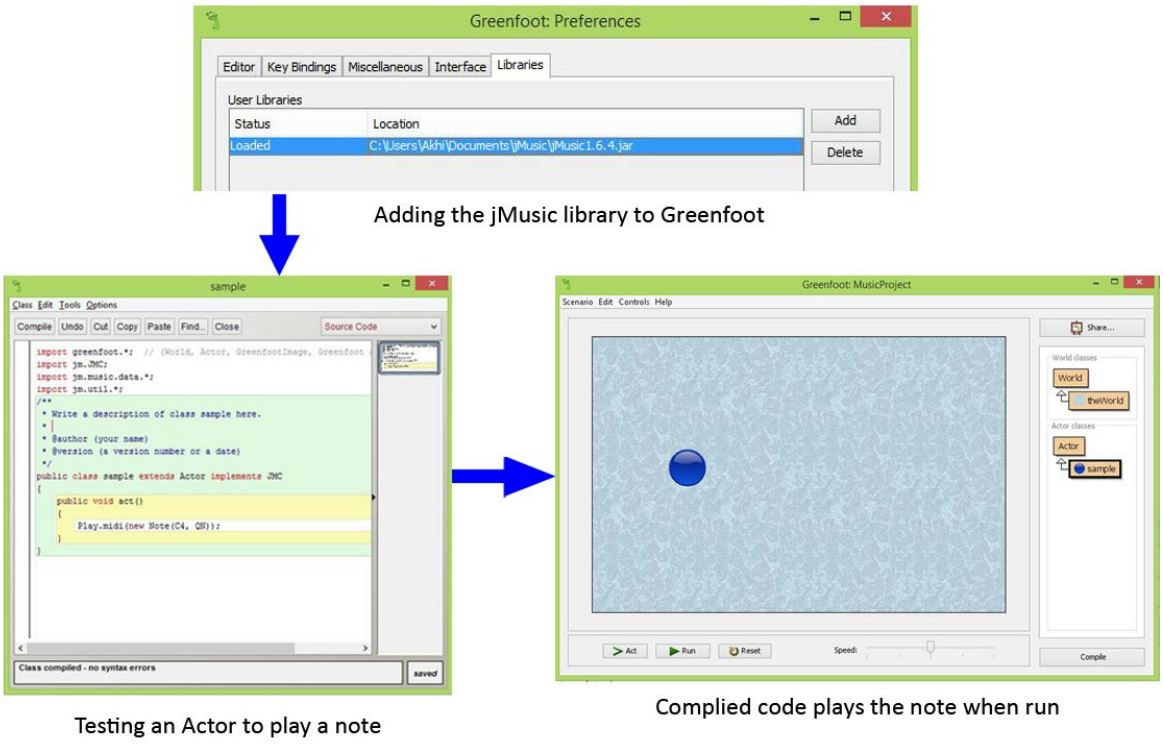
Greenfoot

Greenfoot, using the Java programming language, acts as my main platform for development. This is **justified** because it provides me with a graphical user interface to work with, which is essential for the nature of the software, which requires a visualization of sheet music.

In addition, it is a platform and language that uses **object-oriented programming (OOP).** This is also justifiable by the use of MusicNote objects as the core data structure of the software.



jMusic



I have used jMusic, an external Java library for composing music, and my **justification** is that jMusic allows for an efficient way to play music notes, and is crucial for a large number of the structuring of musical data in the software. An example is in the **playMelody()** method below, which adds a Vector of classes to a jMusic Phrase class, which therefore allows for easy playback.

```
public void playMelody() {
    phraseA.addNoteList(trebleJM, false);
    phraseB.addNoteList(bassJM, false);

    Play.midi(score);
}
```

In addition to using the jMusic library for the composition of music, I also used the help of several tutorials and reference materials from the jMusic creator, Andrew R Brown, to aid in the structure of music data in my software.

Computational Thinking

Thinking Procedurally

The software follows a procedure in the creation of data structures. When a note is added to the staff, it's pitch and image file must be identified, its coordinates calculated, and finally, all of this must be updated to the vectors to be saved for future purposes.

Thinking Logically

I have used logical thinking primarily in the functionality of adding notes to the GUI. For example, when the user selects "C", it is expected that the pitch of the note to be added will be one octave of C from 2 to 5.

The software is also able to logically decide where to place a new note, largely through the use of "if-then" statements. For example, if the selected letter is D and the selected octave is 4, then the X and Y-coordinates of the new note should be placed accordingly.

```
int current = selectors.getPitch(); //The MIDI value of the pitch currently selected.
int spaces = selectors.getSpaces(); //The value of the number of spaces from C. It goes in increments of 5.

if (current >= 72 && current < 84) {
    yPos = yPos5 - spaces;
} else if (current >= 60 && current < 72) {
    yPos = yPos4 - spaces;
} else if (current >= 48 && current < 60) {
    yPos = yPos3 - spaces;
} else if (current >= 36 && current < 48) {
    yPos = yPos2 - spaces;
}

if (isHalfComplete == true) {
    yPos = yPos + 215;
}
return yPos;
```

Thinking Ahead

Prior to the development stage, I thought ahead by using a **top-down** approach to designing the application. In this way, I have created a foundation of necessary functionalities for the software, as well as plans in how to achieve them.

In the development stage, I thought ahead by considering that not every user of this software will input the same number of notes. Because of this, a fixed data structure such as an array is not as suitable as Array Lists or Vectors. The vector will allow a minimum of 8 notes, to a maximum of 64 notes on one musical staff or array.

Thinking Concurrently

Concurrency occurs during the playback of music. Because of the implemented data structure, the music is played back as an entire jMusic Score, which includes all notes in the composition, instead of looping and playing each individual note. This results in a more efficient way of playing the music.

Thinking Abstractly

The data structure for this software requires thinking in an abstract fashion, particularly through the use of jMusic classes, abstracting a piece of music into Notes, Parts, Phrases and Scores.

Word Count: 1190