6.005 Final Design Document

Team Members: Srinidhi Viswanathan, Clare Liu, Yee Ling
clareliu-elainegn-srinidhi

Notable Changes from Original Design:

- No class for Tuplet - We decided that we didn't need a separate class for tuplets because we already calculated the new notelengths of the tuplet in the Listener to find the LCM. Since each note in a tuplet has the same behavior as a regular note, we decided that having a separate Tuplet class would be redundant..
- We decided not to have a Body interface because we decided to deal with measures(bar), notes, tuplets, rests, chords and voices in the Listener. We also made this decision because notes, tuplets, rests, chords and voices are quite different and have different methods. e.g. notes would have a getBaseNote() method, but rest would not and chords would return a list of notes for its getNotes() method.
- No Music Class in body because we found that having voiceList to handle the case where there are voices and cumulativeList to handle the case where there are no voices is sufficient to handle all the music in the abc file.
- Key is not a class but an enum. We have a hashMap (called KeyMap) in Listener that stores all 30 possible keys and maps to a second hashMap that stores the BaseNote (another enum) and its relevant Accidentals for the key given.
- We used the toString() method for comparing equality (we opted out of equals/hashcode) - since we were parsing large abc files, we felt that the toString method did an adequate job of testing for equality.
- No class for Voice, Measure, Repeat - We stored different voices as a list (called voicesList). Each voice was a list of measures(bars), where each measure contains all the notes, rests, tuplets and rests in a bar. For the case where there are no voices, the measure were stored in a cumulativeList and returned. Storing each bar as a measure makes it easy to keep track of the bars where we see [1, [2, or repeat bars or end bars, which makes it easier to implement repeats in the music (see section on Repeats below)
- We did not use LyricListener. We processed Lyrics in the Listener (see section on Lyrics below) and printed them out onto the console with the relevant note.

<u>ABCMusic ANTLR</u>

Tokens:

```
REST : 'z';
DIGIT: [0-9]+;

BASENOTE : 'C' | 'D' | 'E' | 'F' | 'G' | 'A' | 'B'

        | 'c' | 'd' | 'e' | 'f' | 'g' | 'a' | 'b';

TITLE : 'T:' [' ']* [a-zA-Z\.\'0-9()?,' '-/~`!@#$%\^&\*\t\"<>,.:;|\\+={}\[\]]*;
```

```
COMPOSER : 'C:' [' ']* [a-zA-Z\.\'0-9()?,' '-/~`!@#$%\^&\*\t\"<>,.:;|\\+={}\[\]]*;
VOICE : 'V:' [' ']* [a-zA-Z\.\'0-9()?,' '-/~`!@#$%\^&\*\t\"<>,.:;|\\+={}\[\]]*;
LYRIC : 'w:' [' ']* ([a-zA-Z\.\'0-9()?,' '-/~`!@#$%\^&\*\t\"<>,.:;|\\+={}\[\]] | LYRICALELEMENT)* ;
COMMENT: '%' [' ']* [a-zA-Z\.\'0-9()?,' '-/~`!@#$%\^&\*\t\"<>,.:;|\\+={}\[\]]*;
METER: 'M:' [' ']* NOTELSTRICT;
COMMONTIME : 'M:' [' ']* 'C';
CUTTIME : 'M:' [' ']* 'C|';
KEY : 'K:' [' ']* [A-G] KEYACCIDENTAL? MINOR?;

NOTELSTRICT : ([0-9]+'/'[0-9]+);

KEYACCIDENTAL  : '#' | 'b';
MINOR : 'm';
LINEFEED: '\r' | '\r'?'\n';
WHITESPACE : [ \t\r]+ -> skip ;
ACCIDENTAL : '=' |'^' | '^^' | '_' | '__';

TEMPO : NOTELSTRICT '=' DIGIT;
OCTAVE : [''']+ | [',']+;

NOTELENGTH : [0-9]*'/'[0-9]+;
NOTELENGTH2 : [0-9]*'/'+;

TUPLET2 : LEFTP [' ']* '2';
TUPLET3 : LEFTP [' ']* '3';
TUPLET4 : LEFTP [' ']* '4';

LEFTP   : '(';
BAR     : '|';
THINTHIN : '||';
THINTHICK : '|]';
THICKTHIN : '[|';
REPEATL  : '|:';
REPEATR  : ':|';
NTHREPEAT : '[1' | '[2';
LYRICALELEMENT : ' '+ | '-' | '_' | '*' | '~' | '\-' | '|';
```

Parser Rules:

```
abc_tune : abc_header abc_music EOF;
abc_header : field_number comment* field_title other_fields* field_key;
field_number : 'X:' fieldnum end_of_line;
field_title : TITLE end_of_line;
```

```
other_fields : field_composer | field_default_length | field_meter | field_tempo | field_voice |
comment;
field_composer: COMPOSER end_of_line;
field_default_length : 'L:' notelengthstrict end_of_line;
field_meter : (METER | COMMONTIME | CUTTIME) end_of_line;
field_tempo : 'Q:' TEMPO end_of_line;
field_voice : VOICE end_of_line;
field_key : KEY end_of_line;
fieldnum: DIGIT;
temp: DIGIT;
temponote: notelengthstrict;
notelengthstrict: NOTELSTRICT;

abc_music : abc_line+;
abc_line : element* LINEFEED (lyric LINEFEED)? | mid_tune_field | comment;
play_types : note | rest | chord | tuplet_element;
element : play_types | barline | NTHREPEAT | WHITESPACE;

note : pitch (notelength)?;
pitch : (ACCIDENTAL)? BASENOTE (OCTAVE)?;
rest : REST (notelength)?;

notelength: NOTELENGTH2 | NOTELENGTH | DIGIT | NOTELSTRICT;

tuplet_element2 : TUPLET2 note note;
tuplet_element3 : TUPLET3 note note note;
tuplet_element4 : TUPLET4 note note note note;
tuplet_element : tuplet_element2 | tuplet_element3 | tuplet_element4;

chord : '[' note+ ']' (notelength)?;

endbar  : THINTHIN | THINTHICK | THICKTHIN;

repeat  : REPEATL note+ REPEATR;
barline : BAR | endbar | repeatbar;
repeatbar  : REPEATL | REPEATR;

mid_tune_field : field_voice;
comment : COMMENT LINEFEED;
end_of_line : comment | LINEFEED;

lyric : LYRIC;
```

<u>Grammar and Listener Strategy:</u>

General Strategy:
We decided to do most of our note and measure processing in the Listener instead of creating separate classes in our ADT to deal with measures, repeats, and key signatures. We created a stack called *cumulativeList* that contains all of the notes, rests, tuplets, and lyrics that should be played by the SequencePlayer. For example, if some measures should be repeated, these measures will be added into the cumulative list twice. If there are mutliple voices, the cumulative list will have mutliple elements, which can be accessed individually by the SequencePlayer so that they will be played simultaneously.

Header fields:
To avoid conflicts from using overlapping tokens, we decided not to create a separate token for text to use in the grammar rules. Instead, we defined text as
_[a-zA-Z\.\'0-9()?,' '-/~`!@#$%\^&*\t\"<>,.:;|\\+={}\[\]]*_ to match all characters and used this regex within every token that requires text. For example, our COMPOSER token is defined as:

COMPOSER : 'C:' [' ']* [a-zA-Z\.\'0-9(),?' '-]*;

Our grammar rule that identifies a composer field is:

field_composer: COMPOSER end_of_line;

In our Listener, in the exitField_composer method, we used the String.split() method to get the text following the semicolon in the COMPOSER token and to discard the “C:” from the string. We then created a Composer object using this text and added it to a list that keeps track of header elements (*headerList<HeaderElement>*). Similar methods were used for all of the other header fields.

When we exit the header, we use the elements in *headerList* to create an instance of a Header object. If a field is not included in the piece, a default value will be used.

Converting Tokens to PlayType Objects:
In our grammar, a note is composed of a pitch and an optional notelength. Each pitch must contain a basenote and can contain an accidental and/or an octave modifier.

In our exitPitch method, we use a stack (*stack*) that keeps track of the order of the pitches in a piece. To process each pitch, we map each accidental to a unique integer, which will later be mapped to an enum value in Accidental in the exitNote method. We also get the number of octave modifiers to determine the pitch’s octave. Once we have a pitch’s basenote, octave, and accidental, we push the pitch onto the stack.

In our exitNote method, we use a stack (*measureStack*) that pushes notes onto a measure. We pop a pitch off of the pitch stack and process this pitch to create a Note Object. If a note in a piece does not contain a length, we set the length to 1 (*Fraction(1,1)*).

In our exitTuplet method, the notes in the tuplet have already been pushed on to the measure stack. We then modify the lengths of these notes and push them back onto *measureStack*. For example, if the tuplet has 3 notes, we pop off the last 3 notes on the stack and multiply each note's length by ⅔ using our Fraction class. Finally, we push these 3 notes back on the stack in reverse order. We decided to modify the notelengths of notes in a tuplet directly in the listener instead of first creating a Tuplet class because our listener also calculates the least common denominator of all of the notes in a piece and the LCM must be calculated using the modified note lengths.

In our exitChord method, we pop off the notes that compose the chord and use these notes to create a new Chord object. The note length of the chord is defined as the note length of the first note in the chord.

If lyrics are present, they will be added to *measureStack* in exitNote just before the corresponding note. This ensures that each lyric will be displayed at the same time that the note is played.

In exitRest, the length of the rest is processed and then a new Rest object is pushed onto *measureStack*.

Key Signatures and Accidentals
Since the header fields of a piece must be processed before the body of the piece, we decided to get the key signature first and modify the accidentals of notes directly in the listener instead of creating a separate method for identifying and modifying accidentals in our Note class.

In our exitField_key method, we convert each possible input to a Key in the Key enum class. We then store this key signature in a global variable. We also created a HashMap for each key signature that stores each basenote and it's accidental.

In our exitPitch method, we first check if the basenote is in the HashMap of the current key signature. If the basenote is in the HashMap, we modify the current pitch's accidental.

We used a similar method to make sure each accidental in a measure carries through the whole measure. We created another HashMap called *notesPlayed* that keeps track of each accidental that appeared in the current measure. This HashMap is cleared after a barline is encountered, which indicates the end of a measure. For example, if F# was played, (F, Accidental.SHARP) will be added to the HashMap. If another F appears in the measure, a sharp will be added when it is pushed onto the Pitch stack. Using a HashMap allows accidentals to override each other if different accidentals are applied to the same base note.

Repeats
#Note that current list refers to cumulative list for the case where there are no voices and current list refers to the currentVoice in VoiceList for the case where there are voices in this section

Case 1: Only :| is seen. By default the integer startRepeat is set to 0, or it is reset to the bar number after the end of a major section(denoted by || or |]). When :| is see in exitRepeatBar, we set the endRepeat to the current bar number(which is equivalent to the current list size). Then we copy all the measures in the current list of measures from measure[startRepeat] until measure[endRepeat] and add all these measures to the current list of measures.

Case 2: Both |: and :| are seen in the music. In this case, when we exitRepeatBar and see |:, we will take the current list size(current bar number) as the integer startRepeat. When we exitRepeatBar and see :|, then we will take the current bar number (which is equivalent to the current list size) as the endRepeat. Then we copy all the measure in the current list of measures from measure[startRepeat] until measure[endRepeat] and add all these measures to the current list of measures.

Voices vs. No Voices
In both cases, all notes, tuplets, rests and chords are stored in their respective measures. This is so that it will be easy to access and copy particular measures for repeats. The difference between the case where there are voices and where there are no voices is that the different voices have to be stored.

Each measure is an ArrayList that contains all the rests, notes, tuplets and chords in a bar.

So when we have voices, we create a voiceList. In the header, everytime we see a new voice that is not in the voiceList, we add the voice name to the list voiceNames and add a new empty List to voiceList for this voice. When we reach the end of a bar, the entire measure is added to the currentVoice in the voiceList.
e.g. voiceList = [voice1[measure1, measure2], voice2[measure1, measure2]]

When we do not have voices, every measure is stored in the cumulativeList. When we reach the end of a bar, the entire measure is added to the cumulativeList in the voiceList.
e.g. cumulativeList = [measure1, measure2]

Lyrics
Lyrics is taken in as a token of a single String which is then parsed to produce the individual lyrics to be displayed per note played. We first split the lyric string at places with spaces and '-'s.
e.g. Wax-ies Dar-gle => [Wax, ies, Dar, gle]

Then we checked the individual parts of the split string for the symbols ~, *, \-, _ and | as defined in the ABC grammar subset.

Lyrics are indicated by ‘w: ‘. When the lyrics string is split, then there is at least one part with either ‘w:’ or ‘w:something’. To differentiate, we split this part of the string at ‘:’. In the case where ‘w:’, the splitting produces only one part and we ignore it because there is no important lyric information that we can get from the part of the string. In the case of ‘w:something’, we want only ‘something’, so we take the second part of the split.

- means breaks in syllables, _ means to hold a lyrics for an extra note, * means that a note is skipped. For all three cases, no lyrics is output at the next note played. To represent that no lyrics are to be played for the next note, we push an empty string on the lyricStack.

~ means that multiple words are played under one note. Since all words connected with ~ is one part after the lyric string is split, we only replace ~ with whitespace and push the entire string to the lyricStack to be output with the next note.
e.g. apple~pie~apple~hi =>apple pie apple hi

- means that multiple syllables are played under one note. After splitting the parts of the string originally with \- now have \ as the last char in their part.
e.g. a\-b => [a\, b]
Everytime we see a part with \ as the last char (such as ‘a\’ in the example above), we add it to holdOn. We do this until the next part does not have \ as the last char, then we push holdOn onto the lyricStack.

When we see ’|’ the boolean holdLyrics is set to true. While holdLyrics is true, we will not print any lyrics. When we exit a barline, then holdLyrics is reset to false and we continue to print lyrics.

LyricStack is created when we first enter abc_line because the lyrics branch is the last child for abc_line. With LyricStack, every time we want to push a note onto a measure stack, we also pop the corresponding lyric from the lyricStack and push it before the note.

Error Handling
Token errors in the input file will be caught by ABCMusicLexer or ABCMusicParser. A runtime exception will be thrown. (Some are checked by LexerTest as well) e.g. the case where non-basenotes are found in the body of the music will not be matched as tokens and an error will be thrown by the lexer.

Our Listener throws an exception when the final barline is missing. It also throws an exception if a voice is declared in the body, but not initialized in the header. If voices are declared in the header, our Listener will also throw an exception if music in the body is not assigned to a voice.

In addition, our Listener checks for start repeats without a corresponding end repeat and throws a runtime exception with a message

Our Listener also throws a runtime error if only [2 appears without [1 and if [2 appears before [1 and if [1 or [2 appears without the other

Additional abc files:
harry_potter.abc - tests that the meter and default note lengths are calculated correctly if these fields are omitted from the header. Both the meter field and the length fields are omitted. Thus, the meter should be set to 4/4 and the default note length should be set to 1/8

mary_had_a_little_lamb.abc - contains lyrics and a repeat. Tests that the lyrics are displayed again when the song repeats. This piece also tests that lyrics will be displayed correctly when there are multiple voices. For the notes, this particular abc file has different octaves and varying note lengths (1 and 2 primarily)

bonnie_banks.abc - contains a [2 before [1, so a runtime exception that indicates this input error should be thrown

under_the_sea.abc - contains a start repeat (|:) without an end repeat, so a runtime exception should be thrown.

Abstract DataTypes

Interfaces:
PlayType - used for elements that should be added to the sequence player

- Methods:
    - public int addToSequencePlayer(SequencePlayer s, int totalTicks, int lcm);
    - public String toString();

HeaderField - used for types contained within header, such as tempo, meter, index, composer, and title. This was primarily used for ease of readability, so the reader could distinguish between header and body elements.

- Methods:
    - public String toString();

Classes:

HeaderField classes:
- public class Header

- Constructor: public Header (int index, String title, String composer, Fraction noteLength, int tempo, Fraction temponote, Fraction meter)
- Instances of Header are immutable because we need not modify it once it is instantiated, so Header has no setter methods
- Methods:

    - public int getTempo()
    - public Fraction getTempoNote()
    - public Fraction getDefaultNote()
    - public String toString()

- public class Composer implements HeaderField
  - Constructor: public Composer (String composer)
  - Instances of Composer are immutable because we need not modify it once it is instantiated, so Composer has no setter methods
  - Methods:
    - public String getComposer()
    - public String toString()

- public class Index implements HeaderField
  - Constructor: public Index (int index)
  - Instances of Index are immutable because we need not modify it once it is instantiated, so Index has no setter methods
  - Methods:
    - public int getIndex()
    - public String toString()

-public class Length implements HeaderField
  - Constructor: public Length (Fraction length)
  - Instances of Length are immutable because we need not modify it once it is instantiated, so Length has no setter methods
  - Methods:
    - public Fraction getLength()
    - public String toString()

-public class Meter implements HeaderField
  - Constructor: public Meter (Fraction meter)
  - Instances of Meter are immutable because we need not modify it once it is instantiated, so Meter has no setter methods
  - Methods:
    - public Fraction getMeter()
    - public String toString()

-public class Tempo implements HeaderField
  - Constructor: public Tempo (Fraction noteLength, int tempo)
  - Instances of Tempo are immutable because we need not modify it once it is instantiated, so Tempo has no setter methods
  - Methods:
    - public Fraction getNoteLength()

    - public int getTempo()
    - public String toString()

-public class Title implements HeaderField
- Constructor: public Title (String title)
- Instances of Title are immutable because we need not modify it once it is instantiated, so Title has no setter methods
- Methods:
    - public String getTitle()
    - public String toString()

PlayType classes:
- public class Note implements PlayType
- Constructor: public Note(BaseNote noteName, int octavesUp, Fraction noteLength, Accidental accidental)
- Instances of Notes are mutable because we need to modify the length for tuplets
- Methods:
    - public BaseNote getBaseNote()
    - public int convertBaseNoteInt(BaseNote noteName)
    - public int convertAccidentalInt(Accidental accidental)
    - public Fraction getLength()
    - public void setLength(Fraction length)
    - public Pitch getPitch()
    - public int getOctavesUp()
    - public int getTicks(int lcm)
    - public int addToSequencePlayer(SequencePlayer s, int totalTicks, int lcm)

- public class Rest implements PlayType
- Constructor: public Rest(Fraction length)
- Instances of Rests are immutable because we never need to modify the length of a rest, so Rests have no setter methods
- Methods:
    - public Fraction getLength()
    - public int getTicks(int lcm)
    - public int addToSequencePlayer(SequencePlayer s, int totalTicks, int lcm)

- public class Chord implements PlayType
- Constructor: public Chord(List<Note> notes)
- Instances of Chord should be immutable because the notes in a chord should never change and the length of the chord should not be changed either
- Methods:
    - public List<Note> getNotes()
    - public Fraction getChordLength()

        ○ public Fraction getChordLength()
        ○ public int addToSequencePlayer(SequencePlayer s, int totalTicks, int lcm)

- public class Lyric implements PlayType
    - Constructor: public Lyric(String lyrics)
    - Instances of Lyric should be immutable because the text of a lyric should never change
    - Methods:
        - public String getText()
        - public int addToSequencePlayer(SequencePlayer s, int totalTicks, int lcm)

Utility classes:

- public class Fraction
    - This class is used for manipulations with meters and note lengths
    - Constructors:
        - public Fraction(int numerator, int denominator)
        - public Fraction(int numerator) - creates a Fraction if given only a numerator
    - Methods:
        - public int getNumerator()
        - public int getDenominator()
        - public Fraction multiply(Fraction other)
        - public int divide(Fraction other) - This method is used to calculate a modified tempo based on the original tempo and default note length, so all quotients used should be integers
        - public Fraction simplify()
        - public double convertToDecimal() - This method is used for setting default values for the note length, depending on whether the meter has a value less than or greater than 0.75

- public class LCM
    - This class is used to calculate the tick counts for each playable element
    - Constructor: LCM(List<Integer> denominators)
    - Methods:
        - public int getLCM() - gets the LCM for a list of integers
        - public int getLCM2(int a, int b) - helper method that calculates the LCM of 2 integers
        - public int getGCD(int a, int b) - helper method that calculates the GCD of 2 integers

Enum classes:
These were primarily implemented for convenience/readability.
Accidental - represents an accidental (sharp, flat, double sharp, double flat, natural, none)
BaseNote - represents a base note (A, B, C, D, E, F, G)

Key - represents one of the 30 possible key signatures (there is some overlap of the sharps/flats, e.g. A minor and C major)

Testing Strategy:

- **LexerTest:**
  - We tested that all individual components of header (such as composer, title, and index) were lexed into the correct tokens.
  - Then simple tests were conducted within the body, testing different types of notes and playtypes.
  - Finally, we tested combinations of both body and header, making sure to test one complex piece as well.
  - Illegal inputs (like invalid baseNotes and headers) were also tested to ensure that a Runtime exception was thrown.

- **FractionTest:**
  - We tested that the alternate constructor for Fraction that only takes one integer creates a Fraction with a denominator of 1.
  - We tested that all of the Fraction methods (simplify, multiply, divide, convertToDecimal) work properly.
    - For the multiply method, we tested multiplication with both types of constructors.
    - In order to test convertToDecimal, we created the compareDoubles helper method to determine whether two doubles are equal to each other.

- **LCMTest:**
  - We tested a various numbers (including both 0 and 1) and > 2 numbers
    - testing with 0
    - testing with all numbers as 1
    - testing double-digit numbers
- **ListenerTest:**
  - The overall testing strategy was to make sure that as many combinations of header and body are tested as possible.
  - For headers we tested:
    - If both M and L are present, If M is present and L is not, If only L is present and M is not, both M and L are not present.
  - For body we tested:
    - Basic basenotes, a scale besides C, fractional/dotted notes, chord, tuplets (all three forms), accidentals, different pitches, repeats, multiple voices
  - Illegal inputs
    - Invalid ordering of the header fields, errors with repeat bars, more voices than initialized, extra accidentals, etc.
- **SequencePlayerTest:**
  - We tested that the SequencePlayer methods work correctly by converting the notes in

piece1, piece2, and piece3 to SequencePlayer events

- We tested that the play method in Main can take an abc file and play the piece in the file
- We also tested files that contain improper musical constructs (such as a missing end barline) will throw Runtime exceptions and not play the music.