# Design Documentation

6.005 Project 1 – Kulpreet Chilana, Kevin Peng, Anubhav Jain

## Datatypes

(All datatypes contain hashCode(), toString() and equals() methods)

**MusicalPiece(title, composer, meterNumerator, meterDenominator, tempoSpeed,tempoNumerator, tempoDenominator, MusicalPhrase[])**

MusicalPiece is an immutable datatype which is a representation of each abc musical piece we parse. The operations for the datatype are defined below:

**getTicksPerBeat()** - returns the number of ticks per beat
**getTitle()** - returns the title of the Musical Piece
**getComposer()** - returns the composer of the Musical Piece
**getMeterDenominator()** - returns the denominator of the Meter, which represents the note value that represents one beat
**getMeterNumerator()** - returns the numerator of the Meter, which represents the number of beats per measure
**getTempo()** - returns the Tempo of the Musical Piece
**getPhrases()**- returns a deep copy of the MusicalPhrases ArrayList
**playPiece()** – adds all the Notes for the Musical Piece to an instance of SequencePlayer and plays them

**MusicalPhrase(Bar[])**

MusicalPhrase is an immutable datatype which is used to represent different voices and the notes associated with each. It takes an list of Bars as an input.

**getBars()** – returns a deep copy of the ArrayList of Bars in the MusicalPhrase
**getTicks()** - returns the number of ticks per beat
**clone() –** returns a deep copy of the current MusicalPhrase

**Bar(int meterNumerator, int meterDenominator)**

Bar is a mutable datatype which is used to represent measures, and is initialized using the the numerator and denominator of the meter. Stores Notes as an ArrayList and stores length as a double

**addNote()** - adds a Note to the Bar
**getTicks()** - returns the number of ticks per beat
**getNotes()** – returns a deep copy of the ArrayList of Notes in the Bar

**(Interface)Note(length, lyric) = PitchNote(int[] MIDIPitch) or RestNote**

Note is an interface implemented by PitchNote and RestNote. PitchNote represents each note in the musical piece and stores its associated pitch as an integer list. This way, we can have multiple pitch's per note to signify a chord. We use PitchCalculator to determine what

Pitch a note represents in the key signature of the musical piece. A RestNote object will play no sound and have no lyrics associated with it but still have a length.
**getLength()** - returns the length of the Note in beats
**getNumerator()** – returns the numerator of the Note length
**getDenominator()** – returns the denominator of the Note length
**setLyric()** – sets the Lyric element associated with the Note, does nothing for RestNote
**getLyric()** - returns the Lyric associated with the Note
**getNote()** – returns the MIDI Value for all the notes associated with the instance of Note as an array of integers

PitchNote – throws an exception whenever the numerator or denominator is 0 or negative

**Voice(isRepeatOn, isOneTwoRepeat, currentRepeatBar, repeatBars)**
Voice is a helper class for the Listener to represent each of the voices and keep track of them while parsing the ABC music file.
**getRepeatOn()** – returns if repeat is on
**setRepeatOn()** – sets the repeatOn value
**getOneTwoRepeat()** – returns if we are in a one two repeat
**setOneTwoRepeat()** – sets the isOneTwoRepeat value
**getCurrentRepeatBar()** – returns the currentRepeatBar
**setCurrentRepeatBar()** – sets the currentRepeatBar
**getRepeatBars()** – returns the ArrayList of repeatBars
**setRepeatBars()** – sets the ArrayList of repeatBars
**addBars()** – adds Bars to the voice's totalBars
**getVoiceBars()** – returns the voiceBars for this Voice

## Using ANTLR to create the AST

We will use a modified version of the grammar given to us by the BNF file.
We split the header and body into two separate files for greater readability. Then, we removed tokens for certain literals such as DIGIT and TEXT and replaced them with literals inside other tokens instead. For a concrete example, we replaced: "field-number ::= "X:" DIGIT+ end-of-line" with

```
"INUMBER: 'X:'[ \t]*[0-9]+;
field_number: INUMBER end_of_line;"
```

as a rule for the lexer and a rule for the parser. This helped us eliminate the problem of overlapping rules such as TEXT and DIGIT, in which case TEXT would usually hold preference over DIGIT in ANTLR because it would match a longer string.

We also used another lexer and parser specifically for the lyrics in which we used these rules.

```
HYPHEN: '-'[ \t]*;
UNDERSCORE: [_]+[ \t]*;
```

```
SPACES: [ \t]+;
ASTERISK: '*'[ \t]*;
BAR: '|'[ \t]*;
WORD: ('\-'|[~A-Za-z0-9.,'"?\\!&@#$\^();/=\+\]\[])+;
EOL: [\r\n];
lyric: word+ EOL* EOF;
word: (WORD ((SPACES* HYPHEN*) | UNDERSCORE?)) | (ASTERISK) | (BAR);
```

The reason for this is that we want to avoid overlapping rules in our main lexer and parser, but since lyrics can accept almost any type of text, it is much easier to first parse an entire line of lyrics in our main parser and then break it down in a special lyric lexer and parser.

Our listener extends the ABCMusicBaseListener, and we have instance variables for the Header: an ArrayList of MusicalPhrases, an ArrayList of Bars, String title, String composer, String key (key signature), int meterNumerator, int meterDenominator, int tempoSpeed, int lengthNumerator, int lengthDenominator, Boolean hasDefaultLength, and Boolean hasTempo.  We also create mutable instance variables of a HashMap voiceHash which we use to map all the voices to their Voice objects which we will use to create the MusicalPiece. Each voice object has a boolean isRepeatOn , Boolean isOneTwoRepeat, Bar currentRepeatBar, ArrayList<Bar> repeatBars, and ArrayList<Bar> voiceBars. Another mutable instance variable is a Bar currentBar, which represents the current bar we're working in. We also have String currentVoice, which is used to determine which Voice object we're working with. We also create mutable instance variables of Pitch pitch, String baseNote, and int noteNumerator and int noteDenominator which we use to when we create Notes. We create a method, removeWhitespaceAtBeginning, which is used to remove the whitespace at the beginning of any parts of the header. Our Listener also has a getMusicalPiece method which returns a new MusicalPiece that represents the parsed abc file in our ADT. We implement different methods to parse the Header, which store the respective values in the corresponding instance variables. We parse out the meterNumerator and meterDenominator by taking our meter token and storing the numbers to the left of the divide sign as the numerator and everything after the divide sign to be the denominator. In our listener, as we exit each voice, we add that new voice to our HashMap voiceHash.

For the body of the music file, we implement methods for exitAccidental and exitPitch, which modify the mutable instance variable of Pitch and make the appropriate changes as specificed by the Accidental / Pitch trees. We've also implemented a helper class called PitchCalculator which takes the key signature and the base note we're trying to create a Pitch for and returns an appropriate instance of Pitch with the sharps/flats for the key handled. We have an exitNote method which creates a Note object from the pitch variable and then adds it the currentBar instance variable. The exitElement method represents bars and will create a new Bar after adding the currentBar to the ArrayList of bars. The exit element also checks if there are repeats and handles them. The exit multi note method handles chords. The enterTuplet_element resets the tuplet variable to be empty and sets isTuplet to be true. The exitTuplet_element adds the notes in the tuplet to our list of notes. The exitNote_or_rest sets the pitch to be null if we've exited a rest. The enterAbc_line resets the bars in the line to be empty and the exitAbc_line method adds lyrics to the line. The exitAbc_music method adds the last voice to the HashMap voiceHash and then adds phrases from the hashMap to our phrases ArrayList. Finally, these MusicalPhrases are added into our MusicalPiece and this represents the whole ABC file parsed.

## Testing Strategy

We plan on testing the following objects we create: PitchNotes, Rests, Bars, MusicalPhrases, Voices, PitchCalculator, MusicalPiece, Listener

**PitchNotes:**

Our testing strategy includes generating PitchNotes of sharps, flats, regulars, and generating notes in different keys.

a) Test that we get an exception if we try to create a PitchNote with length 0 or negative length
b) Testing creation of a chord, with one note being a sharp and other two being regular notes
c) Testing that a PitchNote of length 1/4 and a lyric associated with it gets stored properly
d) Testing that passing null arguments into a PitchNote throws an exception

**RestNotes:**

Our testing strategy includes generating rests of various lengths and with Lyric events associated with them.

a) Create a rest smaller than a measure
b) Create a rest that is length of a measure
c) Creating a rest that is the length of multiple measures

**Bars:**

Our testing strategy includes generating bars with variable length of PitchNotes and Rests, and also trying to overflow the number of PitchNotes that fit into a bar.

a) Create a bar with enough PitchNotes to represent a measure
b) Create a bar with enough RestNotes to represent a measure
c) Create a bar with more RestNotes/PitchNotes than length of measure
d) Adding a null note to a bar throws an exception
e) Ensures that there is no representation exposure in Bar

**MusicalPhrases:**

Our testing strategy involves creating new MusicalPhrases and adding in a variable number of bars into it.

a) Create a musical phrase with one bar
b) Create a musical phrase with multiple bars
c) Ensures that there is no representation exposure in MusicalPhrases

**MusicalPiece:**

Our testing strategy involves creating new MusicalPieces and checking the getter methods work.

a) Create a basic MusicalPiece datatype
b) Test that each getter of MusicalPiece works from a constructor
c) Ensures that there is no representation exposure in MusicalPiece

**Voices:**

Our testing strategy will involve creating a Voice object and ensuring that all attributes are stored correctly.

a) Creation of a Voice object with constructor

b) Passing in null arguments into the constructor throws an exception
c) Ensure that Voice stores whether or not repeat is stored correctly
d) Ensure that Voice stores whether or not one two repeat is stored correctly
e) Ensure that Voice equality comparison works properly

**PitchCalculator:**

Our testing strategy involves testing equality between different pitches which should be equal in their corresponding Key Signature.
a) Compare two notes in A minor and C key signatures and make sure they're equal
b) Compare a note in G major and E minor and make sure they're the same. We also check that the notes are the same as the original note being transposed up an octave.
c) Our final test case is to compare two notes in G# minor and B major and make sure they're equal.

We will also be testing the creation of bars and notes over repeats, which will be handled by our Listener. During a repeat section, the Listener should go back and create the bars and notes within the repeat section over again and add new objects into the proper MusicalPhrase and Bar classes.