Design Documentation

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

Datatypes

MusicalPiece(title, composer, beat_unit, measure_length, tempo, 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:
getTicks() - returns the number of ticks per beat
getTitle() - returns the title of the Musical Piece
getComposer() - returns the composer of the Musical Piece
getBeatUnit() - returns the denominator of the Meter, which represents the note value that represents one beat
getMeasureLength() - returns the numerator of the Meter, which represents the number of beats per measure
getTempo() - returns the Tempo of the Musical Piece
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 copy of the ArrayList of Bars in the MusicalPhrase
getTicks() - returns the number of ticks per beat

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.
addNote() - adds a Note to the Bar
(Throws exception if sum of note length > bar.size)
getTicks() - returns the number of ticks per beat
getNotes() – returns a copy of the ArrayList of Notes in the Bar

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

Note is an interface implemented by Pitch and Rest. Pitch 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 will store the MIDIPitch mapping in a table. A Rest object will play no sound but still have a length and lyrics associated with it.
getLength() - returns the length of the Note in beats
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

Using ANTLR to create the AST

We will use the grammar given to us by the BNF file. We translated that grammar into ANTLR format and used ANTLR to create the Parser and Lexer. 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 tempoBPM, 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 ArrayList of Bars which we will use to create the MusicalPiece. 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 ArrayList to add Bars to using the HashMap voiceHash. 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.

**************** Ignore everything between stars, only for me *****************
we will walk through the Abstract Syntax Tree and parse the abc file provided, creating a MusicalPiece and storing the Tempo, Meter, Title, and Composer for the file. We will then parse through each note, creating the appropriate PitchNote/Rest objects with the proper pitch and length and then adding them to Bar and MusicalPhrase objects. Each MusicalPhrase represents a voice and we will constantly add Bars to respective MusicalPhrase objects for each voice. Errors will be encountered as we parse the file and appropriately thrown if we have a bar with

more beats than allowed or if we try to create notes with flats and sharps together. Finally, the MusicalPhrases will be added back into the MusicalPiece and then played.
****

Testing Strategy

We plan on testing the following objects we create: PitchNotes, Rests, Bars, MusicalPhrases and we also plan on testing repeats.

PitchNotes:

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

Test Case 1: Our first test case will be testing that we get an error if we ever try to create a PitchNote with a length of 0 or a negative length.

Test Case 2: Our second test case will be testing creation of a PitchNote which is a chord, and one note is a sharp, and the other two are regular notes.

Test Case 3: Our third test case will be testing that a PitchNote's Lyric and length are returned properly using the getLyric and getLength methods.

Rests:

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

Test Case 1: Our first test case will be creating a rest that is smaller than a measure.

Test Case 2: Our second test case will be creating a rest that is the length of a measure.

Test Case 3: Our third test case will be creating a rest that is multiple measures long.

Test Case 4: Our fourth test case will be creating a rest with lyric events associated with them.

Bars:

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

Test Case 1: Our first test will be creating a Bar which is made up of enough PitchNotes to be the length of the Bar.

Test Case 2: Our second test will be creating a Bar which is made up of enough Rests to the length of the Bar

Test Case 3: Our third test case will be creating a Bar which is made up of more Rests / PitchNotes than the length of the Bar and thus should return an error.

MusicalPhrases:

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

Test Case 1: Our first test case will be trying to create a MusicalPhrase with no bars in it, which shouldn't work.

Test Case 2: Our second test case will try to create a MusicalPhrase with one bar.

Test Case 3: Our third test case will try to create a MusicalPhrase with multiple bars.

PitchCalculator:

Our testing strategy involves testing equality between different pitches which should be equal in their corresponding KeySignature.

Test Case 1: Our first test case is to compare two notes in A minor and C key signatures and make sure they're equal, which they should be because A minor and C don't change any of the notes.

Test Case 2: Our second test case is to 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.

Test Case 3: 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 and create the bars and notes within the repeat section over again and add new objects into the proper MusicalPhrase and Bar classes.