Harihar Subramanyam
Ryan Chipman
Danielle Man

6.005 Project 1 – Design Document

Overview:

The result of this project is the ABC Music Player. The input to this player is a string representing the path to the file to be played. The player takes this file name and plays the song, and also outputs the lyrics corresponding to each note as the note is played.

At a high level, the program functions as follows:

The file name is provided. The Java File IO classes read the contents of the file into a character stream. This character stream is fed to the lexer, which produces tokens. The tokens are fed to the parser, which generates an abstract syntax tree. The abstract syntax tree is fed to a visitor, which generates a series of MIDI commands to the provided SequencePlayer. The SequencePlayer produces the sound and lyrics which are observed by the user of the program.

Datatype Definition:

Song: This is the immutable, top-level class that represents each ABC song. This is the output of the parser, and the input to the visitor. It consists of a Header (i.e. the metadata of the ABC file) and a Body (i.e. the musical content of the ABC file).

Header: This is an immutable class which stores the content of an ABC file header. It provides getters for the fields of an ABC header (ex. composer, title, tempo).

Body: This is an immutable class which stores the body of the ABC song (i.e. the music and lyrics). It contains a list of type Voice, which represents each of the "voices" (music + lyrics) in the song.

Voice: This is the interface which defines the basic functionality of constituents of the song (i.e. notes (with lyric), chords, and tuplets). It also has an associated voice name, which is specified in the abc file (ex. "V: test_voice").

Music: This is an interface which represents basic functionality of the constituents of the song (ex. the notes, chords, tuplets). It defines methods for getting the length of the music and a method for copying the piece of Music.

Singable: An interface for associating lyrics with a Music object. Consists of a getSyllable and setSyllable method. It is a sub-interface of Music.

Note: This is an immutable class which implements Singable .This is the basis for the music in the song. It includes fields for keeping track of the letter, accidental, octave, duration, and syllable (i.e. the lyric associated with the note). Like all the other derivatives of Music, it incudes getters/setters and methods for the visitor.

Chord: This is an immutable class which implements Singable. It consists of a list of Note objects, which constitute the chord.

Tuplet: This is an immutable class which implements Singable. It consists of a list of Music objects which constitute the tuplet, and the type (duplet, triplet, quadruplet) of the tuplet.

Rest: This is an immutable class which implements Music. It consists of the duration of the rest.

AccidentalEnum: An enumeration which includes the sharp, flat, double sharp, double flat, natural, and none (i.e. no accidental applied).

KeySignature: An class which returns the array of accidental offsets for a given key signature (by looking them up in a pregenerated hashmap).

NoteEnum: An enumeration of the letter notes (i.e. A, B, C, D, E, F, G). Each note is associated with a pitchScale (according to the values defined in the SequencePlayer).

TupletEnum: An enumeration of the tuplet types (duplet, triplet, quadruplet).

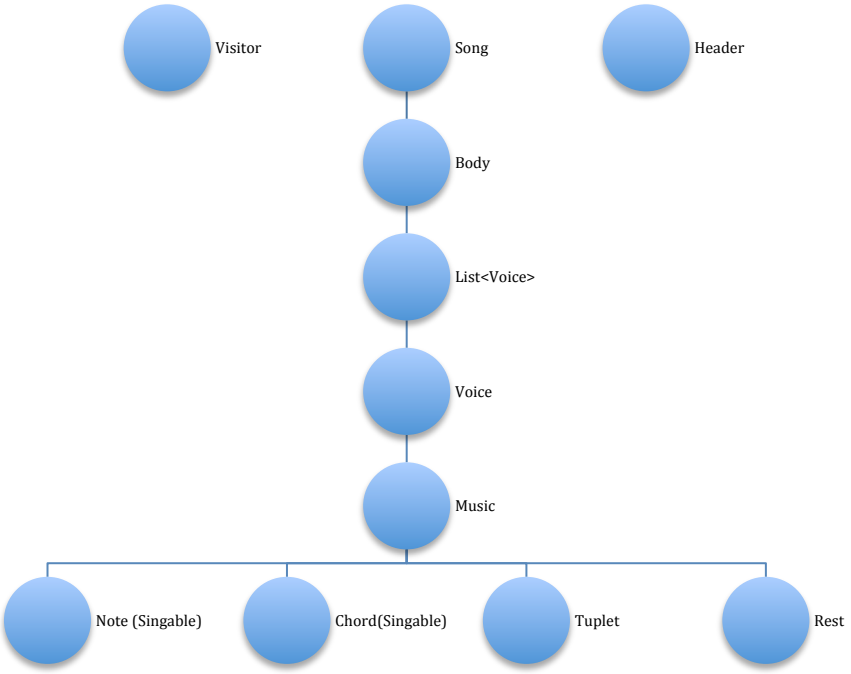
Fraction: An immutable class for representing rational numbers (by storing their numerator and denominator). Includes a least common multiple method. This class is used to compute the ticks per beat needed for the SequencePlayer.

ISongSequencerVisitable: The "Visitable" interface as described by the Visitor pattern. Contains an accept method. Song, Header, Body, Voice, and all Music derivatives implement this interface.

ISongSequencerVisitor: The "Visitor" interface as described by the Visitor pattern. Contains methods for visiting all objects contained in a Song.

SongSequencerVisitor: Class which implements ISongSequencerVisitor. This is a mutable class which takes a Song object and recursively walks down the tree, adding notes and lyrics to the SequencePlayer. It uses the tempo, meter, and default note length information from the header to appropriately sequence notes with the correct durations, key signatures, etc.

SongListener: Class which generates a Song object from the abstract syntax tree produced by the parser. A substantial volume of code lies within this class. The class is responsible for extracting information from the header, creating notes (with appropriate lengths, accidentals, octaves, etc.) from their strings, and grouping notes into tuplets and chords appropriately.



Grammar:

The grammar has been designed to minimize the number of lexer token definitions and parser rules. The lexer includes tokens for each of the header fields, basic music elements (ex. note, pitch, accidental, rest, barline, tuplets), and fundamental symbols (ex. brackets, digits, whitespace).

The parser consists of rules to create the header and rules to define the lines of the song. The rules were designed and modified to reflect the needs of the listener.

Using ANTLR:

The parser takes the lexer tokens and generates an abstract syntax tree. The listener includes a number of methods to walk recursively down the tree and generate a Song object. Most of the heavy lifting is done in the listener.

Abstract Data Type to sounds:

Given the Song object (recall that this is the highest level of our abstract data type), we use the Visitor pattern to walk through the constituents The Visitor begins with the Song object, extracts relevant information from the header (ex. key signature, tempo) and recursively walks through the constituents of Song and each of its subclasses and visits them as well. The result of the Visitor is a set of objects which can be added to the SequencePlayer and played.

Testing:

We want to test as many components as we can. We began by testing the warmup exercises. We test the lexer, ensuring that it can tokenize all the characters that we described in the grammar and crashes on invalid inputs. The parser+listener tests include a test that ensures that the parser can parse a song that makes use of every feature of the ABC subset (ex. multiple voices, tuplets, chords). The Visitor will be tested by giving it a Song object which utilizes all the different possible constituents (ex. lyrics, multiple voices, key signature) and ensuring that it generates the proper sequence of pitches and lyrics for the SequencePlayer. Finally, an overall test was written for the entire player – feeding it all the sample files and ensuring that it can play all the sounds. We also found four abc songs from the internet, which we included in our player's sample files.

Changes from Initial Design:

The biggest change we made was to the grammar. At first, we defined TONS of small, simple lexer tokens and very TONS of small, simple parser rules. We reasoned that by keeping the rules simple, it would be simpler to write code to process them. However, we found out that the problem with this approach was that the rules were too simple, and often resulted in incomplete structures which the Listener could not easily process (ex. what should we do with a note that has a letter and duration, but is missing its accidental and octave – not a good design, especially with immutable classes). In our final design, we minimized the number of tokens, and made the parser rules more sophisticated, which allowed us to write only a few methods in the listener, instead of several. Thinking of the English language, this made sense – a few "parts of speech", with sophisticated ways to combine them.

The other change we made was to distribute work among the several classes. Our initial design effectively required the parser to do all of the heavy lifting, which made it incredibly unwieldy and hard to debug. In our final version, we moved some of this responsibility to the listener and visitor, thereby making the work evenly distributed. This also had the advantage that we could more easily test the separate components with simple tests, instead of testing the "super parser" with a large number of complicated tests.