Harihar Subramanyam
Ryan Chipman
Danielle Man

6.005 Project 1 – Design Document

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). It provides getters for the Header and Body, and also implements convenience functions which the Visitor can make use of (ex. getKeySignature(), getTempo()).

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). It also implements functions for the Visitor to use.

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. It provides getters for the fields and relevant methods for the Visitor.

Voice: This is the interface which defines the basic functionality of constituents of the song (i.e. music + lyrics). It provides methods for the Visitor to use this object to generate a sound which can be played by the SequencePlayer (i.e. how many ticks is it? When is it played?)

Music: This is an abstract class which represents the non-lyric parts of the song (ex. the notes, chords, tuplets). It defines methods for getting the length of the music, Visitor related methods, a method to convert the music to a sound that the SequencePlayer can play.

Note: This is an immutable class which inherits from Music and implements Voice. This is the basis for the music in the song. It includes fields and methods for the Visitor method to generate a sound that the SequencePlayer can play. For instance, it provides methods for retrieving the note type, length, octave, and accidental.

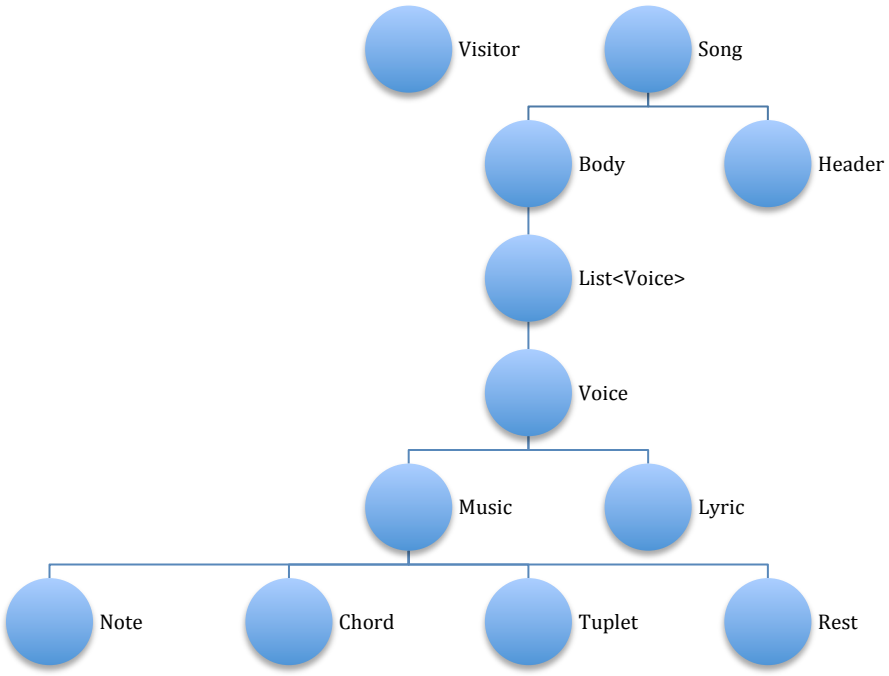
Chord: This is an immutable class which inherits from Music and implements Voice. It provides the standard getters and provides methods for the Visitor to use. Furthermore, the chord is represented by a list of Note objects.

Tuplet: This is an immutable class which inherits from Music and implements Voice. It provides the standard getters and provides methods for the Visitor to user. Furthermore, the Tuplet is represented by a list of Music objects (we do not use a list of Note objects because a tuplet can contain chords).

Rest: This is an immutable class which inherits from Music and implements Voice. It provides usual basic functions like toString, equals, and hashCode. The class provides a method to get the length of the rest, so the Visitor (and therefore the SequencePlayer) will stay silent for the duration of the test.

Lyric: This is an immutable class which implements Voice. It provides methods and fields for the Visitor and the SequencePlayer – and it includes the actual lyrics which are sung.

Visitor: This is a mutable class which takes a Song object and recursively walks down the tree, building itself up and finally producing a list of pitch, lyric, and timing information which the SequencePlayer can play.



Grammar:

The grammar has not been changed. The reason for this is because the ADT has been designed around the grammar, which thereby minimizes the lexing and parsing work required, and also does not require the grammar to change (as far as we know).

Using ANTLR:

The parser uses the parse tree and builds a Song object according to the datatype definitions. As we travel down the parse tree, we can build up the ADT tree. Since the ADT is built as a tree structure and is heavily influenced by the grammar, it should be easier to generate the ADT from the parse tree.

Abstract Data Type to sounds:

Given the Song object (recall that this is the highest level of our abstract data type), we will use the Visitor pattern to produce a list of Pitch objects, Lyric strings, and the associated timing duration. 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. Next, we'll test the Lexer, ensuring that it can tokenize all the characters that we described in the grammar and crashes on invalid inputs. The Parser tests will 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, repeats). Then, invalid constructs, missing parameters, and poorly formatted expressions will be tested to ensure that the parser cannot parse them. The empty string will be tested to ensure that the parser returns an error. 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. The abstract data type's functions, like toString, equals, and hashCode will be tested for proper relationships (ex. reflexive, symmetric). Utility classes, like Fraction class, will be tested by providing inputs for all operations and checking edge cases (ex. zero numerator, zero denominator, etc).