­­Harihar Subramanyam

Ryan Chipman

Danielle Man

# 6.005 Project 1 – Design Document

Overview:

Our abcplayer consists of a Lexer, a Parser, and a Player. The player is aided by a complete datatype structure that we refer to as our ADT. The Lexer provides a list of tokens recognized from an inputted abc file. The Parser takes this list of tokens and transforms them into a complete Song, comprised of the datatypes that we defined in our ADT. The song is walked by our Visitor which transforms our ADT’s components in to pitch, lyric, and timing information that can be understood by the MIDI SequencePlayer. We use this sequencer to create sound and play the song with lyrics being displayed on the system according to the rhythm of the music.

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, rests). 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.

**Singable:** This is an abstract class that represents the lyrics in a song. Singable extends music and contains getSyllable() and setSyllable() methods. Note and Chord both implement Syllable and it is these two methods that allow us to associate syllables with notes and chords respectively to ultimately display lyrics. Our listener sets syllables to notes and chords when we first parse the abc file and create the song. Then our visitor gets the syllables from the notes that it is playing and adds them to the queue of things for the SequencePlayer to 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.

Other Classes in Player:

We have defined several different Enums.

**AccidentalEnum** – Specifies the different types of accidentals we can have in abc notation (sharp, double sharp, flat, double flat, natural).

**NoteEnum** – Specifies that notes can only be C,D,E,F,G,A, and B.

**TupleEnum** – Specifies that tuples can only take the form Duplet, Triplet, and Quadruplet.

**KeySignature:** This provides a map of accidentals associated with a particular key signature. This is used in the listener when we first create our Song object because given a particular key signature (other than C major or A minor), we will need to apply accidentals on certain notes automatically, without the abc notation specifying it.

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. This contains visit() methods for Song, Header, Body, and Voice, but the main method is play().

Play starts by creating a new SequencePlayer with information that it is given in the header of the song (ex. tempo). It sets the ticks per beat parameter by calculating the least common multiple of the different note durations throughout the piece. After initializing the SequencePlayer the next step is to add the notes. Play keeps a running count of the start tick and the tick duration, the two timing variables that are important when adding notes to the sequencer. Play runs through the list of Musics that we passed it and checks to see if they are instances of Chord, Tuple, Rest, or Note. Play then handles those different cases accordingly (adds Notes to the sequencer with the correctly paired timing information with the .add() method), incrementing the start tick count with each new element in the Musics list. The tick duration is calculated individually for each Music element. We multiply the ticks per beat (calculated when we initialized the sequencer) by the note duration (extracted from the individual Music elements) to acquire this number. With each call of .add() we build up a list of music to be played and ultimately we create sound with a .play() call to our SequencePlayer.

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).

Notes on the Lexer and Parser:

Our lexer creates tokens that are then recognized by our parser and sent to the listener. The listener builds up a Song object consisting of notes, rests, chords, and tuples at the base level, and a body and a header at the highest level. Our complete song object is passed in to the visitor which has a play() method that will transforms our Song object in to notation that can be played by the MIDI SequencePlayer.

Notes on changes that we’ve made since the Milestone: