Design Document

Table of contents:

- Data Types
- Dependency Diagram
- Parsing Strategy
- Conversion Process
- Testing Strategy

Link for the editable online version of this document

Data Types

Duration (Immutable)
A reduced fraction n/d representing the duration of a musical note. For example, an eighth note would have n=1 and d=8. Invariant: n and d are coprime. n and d are positive and non-zero.

Operations:

- constructor(numerator,denominator)
- getNumerator(): return the numerator (n) of the fraction
- getDenominator(): return the denominator (d) of the fraction
- add(duration): return a new duration consisting of this duration plus another duration
- mul(duration): return a new duration consisting of this duration times another duration
- div(duration): return a new duration consisting of this duration divided by another duration
- equals(other):
- hashCode():

Accidental (Enum)
A data type representing a sharp, a flat, double sharp, double flat, natural, or none.

Operations:

- N/A

SingleAccidental (Enum)
A data type representing a sharp, a flat, or natural.

Operations:

- N/A

Letter (Enum)

A data type representing the musical letters “A” through “G”

Operations:

- N/A
- fromChar(c): static function to parse a single-character to a letter object

Pitch (Immutable)
A modification of the provided “Pitch” class for increased type safety (static checking via Enums) and more simplicity (the “Pitch” class in the handout code was absolute garbage).

Operations:

- getLetter(): return letter
- getAccidental(): return accidental
- getOctave(): return octave
- toMidiNote(): return the midi note of the pitch
- equals(other): two pitches are equal if their letters, accidentals, and octaves are equal
- hashCode(): something reasonable

KeySignature (Immutable)
A mapping from notes to accidentals based on the musical notion of key signature.

Operations:

- constructor(letter, accidental, major): construct a key signature with a tonic note given by the letter and accidental (single accidental), using a major/minor scale based on the boolean “major” flag
- getAccidental(letter): return the accidental for the given letter in this key signature

Sound (Immutable Interface)
A list of pitches associated with a sound. A “note” contains a single note, a “chord” contains multiple note, and a “rest” contains no notes.

Operations:

- getPitches(): return an unmodifiable list of Pitches
- equals(other): returns if the other sound has the same pitches (including accidentals)
- hashCode(): something reasonable

Note (Implements Sound)
A single pitch.

Operations:

- constructor(pitch): create a note
- getPitches(): return an unmodifiable list containing a single Pitch

Chord (Implements Sound)
A list of more than one pitch.

Operations:
- constructor(pitches): create a chord with the given non-empty list of pitches
- getPitches(): return an unmodifiable list of Pitches

Rest (Implements Sound)
No pitches.

Operations:
- constructor(): create a rest object
- getPitches(): return an unmodifiable tmpty list of Pitches

SoundEvent (Immutable)
A Sound associated with a duration. A “note” contains a single note, a “chord” contains multiple note, and a “rest” contains no note.

The duration of the sound event is the relative duration of the note. To get the absolute duration of an event, multiply its relative duration with the defaultDuration of the song.

The pitches of a sound event do not always reflect the true pitches of the notes that are to be played back. In particular, pitch changes due to key signature and previous accidentals in the bar are not factored into the pitch of the current note.

Tuplets not given any special attention in the AST. The parser should expand a tuplet into its individual notes, and then make individual SoundEvents for the notes.

Operations:
- constructor(sound, duration): create an event with the given arguments
- getSound(): return a Sound object
- getDuration(): return the duration
- equals():
- hashCode()

Lyric (Immutable)
A string of characters corresponding to a single syllable of song lyrics.

Operations:
- constructor(text): the given text (potentially empty) is associated with a single syllable
- getText(): return text

RepeatEnding (Enum)

An enum indicating whether a bar marks the beginning of the first ending ‘[1’ (FIRST), the second ending '[2’ (SECOND), or none at all (NONE).

Operations

- Not applicable

Bar (Immutable)
A group of one or more SoundEvents and Lyrics that make up single musical bar.

There are different types of bars in the music. Some bars are the beginning of repeated sections. Some bars end repeated sections. Some bars indicate the beginning of a first repeat ending (eg. “[1”). Some bars are just normal bars

To classify the different types of bars, we use the following variables:

- beginRepeat (type boolean)
- endRepeat (type boolean)
- repeatEnding (type repeatEnding)

Operations:

- constructor(events, lyrics, beginRepeat, endRepeat, repeatEnding): creates a bar with the given parameters
- getEvents(): return unmodifiable list of events
- getLyrics(): return unmodifiable list of lyrics
- getBeginRepeat(): return beginRepeat
- getEndRepeat(): return endRepeat
- getRepeatEnding(): return repeatEnding

Voice (Immutable)
A unique identifier for representing musical voices. A voice with a blank name corresponds to the “default” voice in a song.

Operations

- constructor(name): initialized the voice with the given string name
- equals(other): two voices are equal if they have the same name
- hashCode(): implemented appropriately

Song (Immutable)
Sequences of Bar objects organized by Voice, and some metadata, including its index, title, composer (C), meter numerator (M), meter denominator (M), default duration (L), key signature (K), beat duration (Q), and beats per minute (Q).

Conceptually, the Song contains a Map from Voices to List<Bar> objects. The parser is expected to construct this map externally, and then pass it to the Song constructor as the first

“barLists” argument.

If no voice name is specified in the abc file, then the default Voice object is used.

Operations:

- constructor(barLists, index, title, composer, meterNumerator, meterDenominator, defaultDuration, keySignature, tempoDuration, beatsPerMinute): constructs a song with the given parameters
- getSections(): return unmodifiable list of sections
- getIndex(): return index
- getTitle(): return title
- getComposer(): return composer
- getMeterNumerator(): return meterNumerator
- getMeterDenominator: return meterDenominator
- getDefaultDuration(): return defaultDuration
- getKeySignature(): return keySignature
- getBeatDuration(): return beatDuration
- getBeatsPerMinute(): return beatsPerMinute
- listVoices(): return List<Voice> of the names of the voices
- getBars(voice): return List<Bar> of the bar objects for a given voice

PlayableSoundEvent (Immutable)
A Sound object, the duration of ticks, and an optional Lyric string. This represents a single note/rest/chord which can be played by the SequencePlayer.

Unlike the SoundEvent object, the pitch of the sounds in PlayableSoundEvent are exact (they take into account the key signature and other accidentals).

Operations:

- constructor(sound, ticks, lyric): create object with lyric
- constructor(sound, ticks): create object without lyric
- hasLyric(): return true if object has lyric
- getLyric(): return lyric
- getSound(): return sound
- getTicks(): return ticks

PlayableSong (Immutable)
A list of PlayableSoundEvent objects which are to be played in sequence at a particular beatsPerMinute and ticksPerBeat.

Operations:

- constructor(events, beatsPerMinute, ticksPerBeat)
- getEvents(): return events

- getBeatsPerMinute(): return beatsPerMinute
- getTicksPerBeat(): return ticksPerBeat

SongParser (Immutable)
Parse the given abc file into a song object.

Operations:
- constructor(filecontents): parse the given filecontents string into a Song object
- getResult(): return the song object

SongConverter (Immutable)
Converts the given Song object into a PlayableSong object, and checks invariants of the resulting song object (eg. all sequences within a bar have the same length. All bars have the same length)

Operations:
- constructor(song): convert the given Song into a PlayableSong
- getResult(): return the playableSong object

SongPlayer (Immutable)
Plays the given PlayableSong object. Print lyrics using the given LyricListener.

Operations:
- constructor(playableSong, LyricListener): initialize with these parameters
- play(): play the song

LyricListener (Immutable, Interface)
Outputs lyrics of a song in an implementation defined manner.

Operations:
- procesLyricEvent(text): output the given lyric string

ConsoleLyricListener (Implements LyricListener)
Prints lyrics to the console.

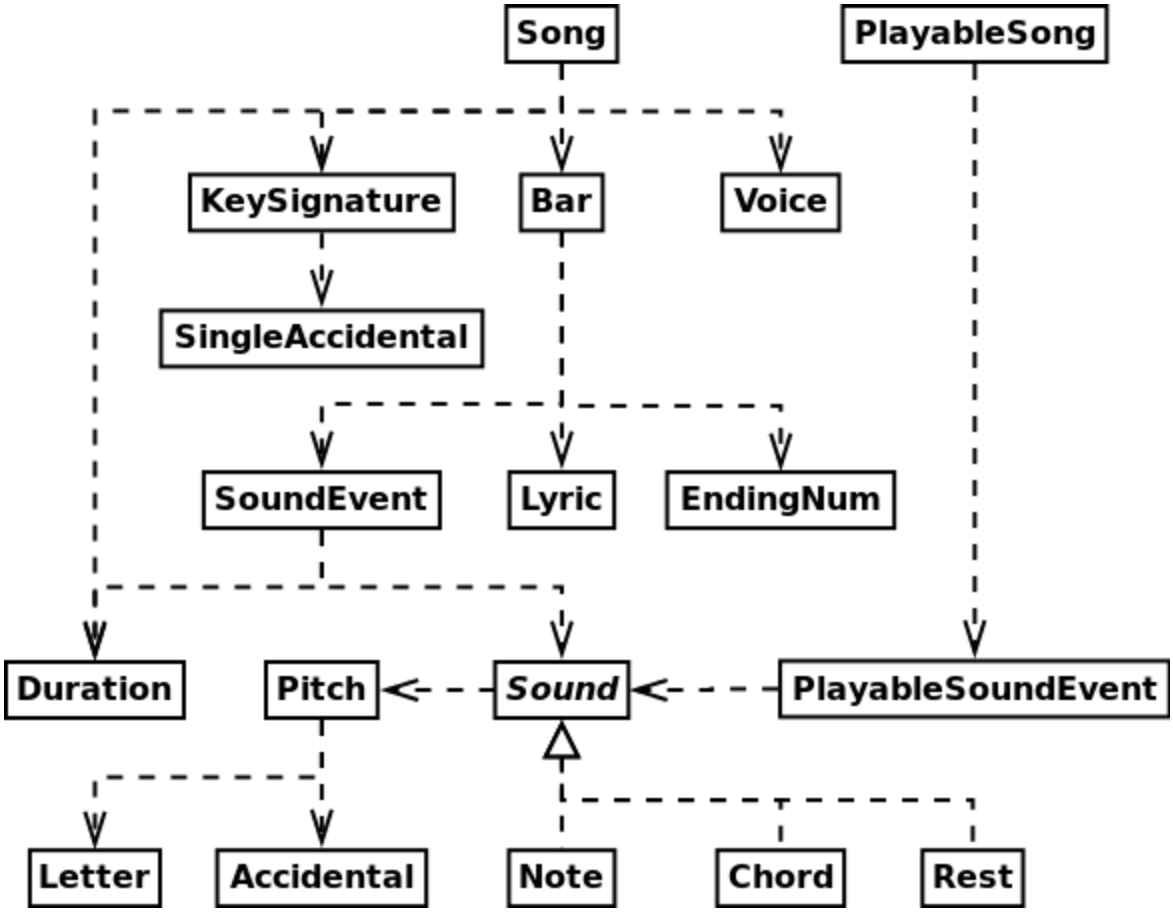
Operations:
- processLyricEvent(text): prints text to the console

SequencePlayer
Interfaces with the Java midi libraries. From the handout code.

Operations:
- provided in the handout code

Dependency Diagram

Dependency diagram of the Abstract Syntax Tree and other data types. Classes that don't encapsulate the music data (such as SequencePlayer and LyricListener) are not included in this diagram.



The source for this diagram is located in doc/dependency_diagram.dia, and it can be edited using the free, open source editor called "Dia".

Parsing Strategy

We use Antlr to define the grammar. The SongParser class will contain the necessary logic to convert an abc-formatted file into a Song object (the AST).

The AST was designed to match the abc specification as closely as possible. This design was chosen to minimize the amount of state in the parser while walking the parse tree.

For example, the note lengths and accidentals are transformed into SoundEvent objects exactly as they are written in the abc file. The true lengths and pitches of these notes (taking into account the bpm, default note length, key, and previous accidentals occurring in the same bar) are calculated after the parsing is complete.

Similarly, the bar-repetitions symbols are stored in the Bar object of the abstract syntax tree when the parse tree is walked, but the validation and implementation of repeated bars occurs after parsing is complete.

There are, however, a few cases where the parser needs to keep track of state.

First is the case of tuples. Instead of storing a “Tuple” object in the AST, the parser converts a tuple into individual SoundEvents while walking the tree, and inserts them into the AST. This was done to sidestep the complexity of introducing a separate “Tuple” object into the AST. (Remember to throw an error if a rest is part of a tuple)

Second is the case of chords. When a single chord contains notes of different lengths, the duration of the first note written in the chord is the duration of the entire chord. (The specification is ambiguous about what to do with the durations of the subsequent notes. So, we will ignore them for now.) When two durations are provided like “[C2E]4” the durations are multiplied, resulting in “[CE]8”. All of this math occurs while the parser is walking the parse tree.

Third is the case of Lyrics. According to the abc specification, Lyrics may only occur on the line following the notes. Rather than introducing a “Line” object into the AST, it is simpler to store the number of bars in the previous line, and augment the previously-created Bar objects with lyrics when lyrics are read on the current line.

Last is the case of Voices. Because the abc file format allows the voices to be interleaved, the parser must keep track of the last seen Voice tag so that it can match newly created Bar objects with the proper voices. In this case, the easiest solution is to maintain a mapping from Voice objects to List<Bar> objects and have an instance variable that tracks the last-seen Voice name. Whenever a new Bar object is parsed and created, it is appended to the list corresponding to the last-seen Voice.

After the parser has completed, it runs a validation script to enforce any additional constraints that are not enforced by the Antlr grammar, and raising RuntimeExceptions if they are violated. This includes ensuring that:

- all bars have the same length.
- repeat bars are valid

Conversion Process

The output of the parser cannot be played directly into the MIDI synth. As noted in the previous section, accidentals, key signatures, tempo, and repeated bars must be handled first. The purpose of the conversion process is to handle these special cases.

The output of the SongParser class is an abstract parse tree represented by a Song object. This Song object gets fed into a SongConverter object to produce a PlayableSong object. The PlayableSong object contains the song data in a flat, list-like format so that it can be easily converted into MIDI by the SongPlayer class.

Following is a list of special case the SongConverter must handle, as well as the strategies used to solve them.

- Not all accidentals are present in the AST. Missing accidentals must be filled in based upon (1) the key signature stored in the header, and (2) previous accidentals within the same bar. We used the following algorithm to fill in the missing accidentals:
    - for each bar
        - for each letter:
            - map[letter] = key_signature[letter]
        - for each note (including ones in chords):
            - if note.accidental == Accidental.NONE
                - note.accidental = map[note.letter]
            - else
                - map[note.letter] = note.accidental
- The SongParser ensures that the bar repetitions are well-formed. Therefore, the algorithm for generating repeating bar segments is extremely straightforward.
- Lyrics must be aligned with the notes. Secondly, the list Bars must be flattened into a list of PlayableSoundEvents. Lastly, the duration of the notes must be converted into ticks. Its easiest if these three steps are combined.
    - gcd = the LCM of {event.duration.mul(song.defaultDuration).denominator | all events in all bars} union {song.beatDuration.denominator}
    - beatsPerMinute = song.beatsPerMinute
    - ticksPerBeat = gcd/song.beatDuration.denominator*song.beatDuration.numerator
    - list = []
    - for each bar
        - lyric_index = 0
        - for each event
            - absoluteDuration = event.duration.mul(song.defaultDuration)
            - numTicks = absoluteDuration.numerator * gcd / absoluteDuration.denominator
            - text = null
            - if lyric_index < bar.lyrics.length

            - text = bar.lyrics[lyric_index].text
            - lyric_index += 1
        - sound = event.sound
        - if text
            - list.append(new PlayableSoundEvent(sound, numTicks, text))
        - else
            - list.append(new PlayableSoundEvent(sound, numTicks))
    - return new PlayableSong(list, beatsPerMinute, ticksPerBeat)

Testing Strategy

Our design has several distinct modules that are well-suited for testing. Following are descriptions of these modules, and our strategies for testing them.

- **SongParser**
    - Description: parses and validates the abc file
    - Testing strategies and examples:
        - unit tests for the header
            - header elements may appear in any order
            - all elements except for X, T, and K are optional
            - missing X, T, or K should raise exceptions
            - the M, L, Q, and C headers have defaults if they are not included
        - unit tests for the body
            - all octave identifiers are correctly interpreted (capitals, lowercase, commas, and apostrophes)
            - all relative note length formats are correctly parsed (‘2’, ‘2/3’, ‘/3’)
            - all accidentals are correctly parsed (‘^^’,’^’,’=’,’_’,’__’)
            - a rest ‘z’ is correctly parsed with duration
            - chord durations are correctly parsed (first inner duration has precedence), and multiplied together when necessary (‘[C2E1]4’ has relative duration 8)
            - durations of duplets, triplets, and quadruplets are correctly calculated
            - the various repeat symbols set the proper flags on the ‘bar’ class
        - unit tests for Lyrics
            - lyrics get associated with the proper bar when the previous line is 1 or greater than 1 bars long
            - “~” “_” “*” “-” “\-” symbols are properly handled
        - unit tests for Voices
            - the default voice is used when none is provided
            - multiple voices are supported

                - lyrics get associated with the proper voice
                - bar repeats/endings get associated with the proper voice
            - unit tests for AST validation
                - raise error if not all bars are the same length
                - raise error if repeat bars are invalid (two '|:' in a row, two ':|' in a row unless they are separated by the end of a major section '|]', a '|:' without a closing ':|', a '|[1' which is not followed by a ':|[2')
- **SongConverter**
    - Description: converts Song objects to PlayableSong objects
    - Testing strategies and examples
        - unit tests for accidentals
            - accidentals are filled by the key
            - accidentals affect later notes in the bar
            - accidentals don't affect notes in later bars
            - accidentals override each other within the same bar (eg,"natural" overrides the previous state)
            - accidentals are filled in with chords
        - unit tests for bar repetition
            - '|:' ':|' works
            - ':|' automatically repeats from the beginning of the song
            - '|]' ':|' automatically repeats from the beginning of the last major section.
            - '|:' '|[1' ':|[2' works as expected
        - unit tests for lyrics
            - multi-word lyrics get aligned with the proper notes
            - bars with too many lyrics truncate the extra lyrics
            - bars with too few lyrics don't assign lyrics to the final notes in the bar
        - unit tests for duration
            - beats per minute, ticks per minute, and note duration should be calculated properly
- **SongPlayer**
    - Description: pipes a PlayableSong into the SequencePlayer and LyricListener
    - Testing strategy and examples
        - general tests
            - midi should be sent to the SequencePlayer when song is playing
            - lyrics should be sent to the LyricListener when song is playing
            - beatsPerMinute and ticksPerMinute are passed to the SequencePlayer