ABC Player Design Notes

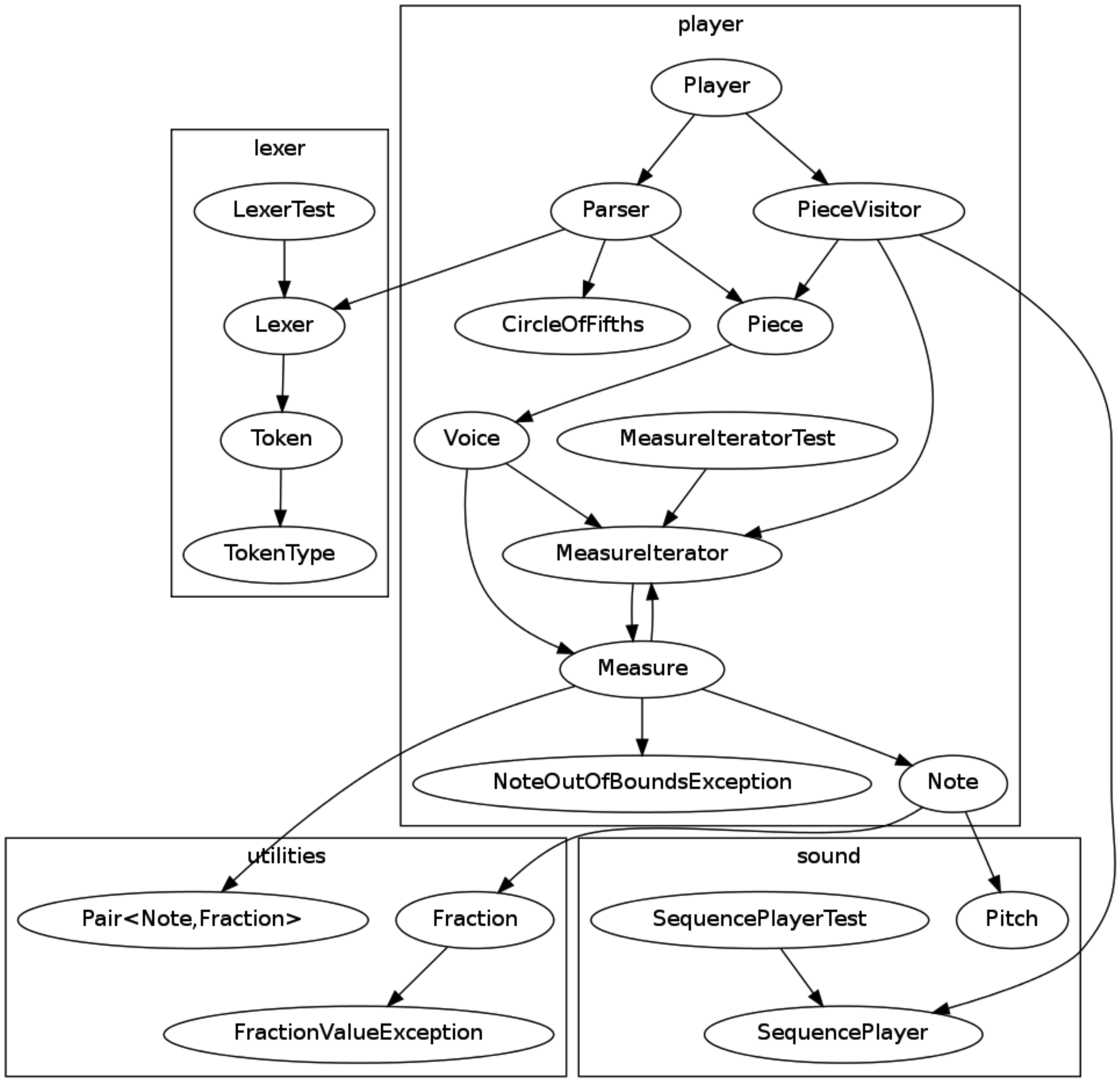
Catherine Zuo, Kimberly Toy, Will Oursler

Contents

The following document addresses the following:

1. Dependency Diagram

The following dependency diagram describes all of the classes that will be included in the ABC music player. The boxes represent classes, which form common functionalities.



2. Design Flow

Lexing

TokenType objects describe the type of tokens that can be made (i.e. note or accidental tokens) while Token objects represent the different kinds of tokens that are lexed from the .abc file contents.

A Lexer object is created with a list of TokenTypes representing the kinds of tokens that it can lex. It accepts the .abc text input and returns a list of valid tokens or throws an exception if invalid tokens are found.

Parsing

The Parser object uses the Lexer object to get the tokenized version of the music file and creates a Piece object, which is our representation of the .abc input.

The Parser will use recursive descent parsing. It will look ahead in the list of tokens and determine from the token types which production to use. For example, when encountering a field header token “T:”, the parser will recognize that this and the text tokens before a line break and know that this represents a field-title as described in the grammar. In terms of parsing the musical body, the parser will look for an element token (i.e. a note-element, tuple-element, etc.) and parse the following tokens before the line break as a abc-line.

The Parser will track and throw errors when encountering invalid measure durations (too long or too short), incorrect repeat braces, musically incorrect accidental stacking, invalid tuplets (length-wise), and invalid beat times (negatives, decimals, and the Fraction class should ensure there are no divide by 0s). It will catch these errors through the nature of recursive descent parsing; for example, opening and closing repeat braces are expected to match correctly in number, and the parser will throw an exception if there are no more tokens to parse when there is an unclosed repeat brace, or if there is an matching closing brace. Invalid lengths of measures will be caught by keeping a running total of the measure length that is being created and throwing an exception when extra beats are added.

Creating a Piece

The Piece is our musical representation data type, which is created as the parser parses the .abc input tokens. It contains fields, which describe the piece data as described in the .abc file headers, such as tempo, default length of a note, and meter. The Piece object contains references to the different voices in the piece.

A voice object represents the musical lines that belong to one voice in the abc file. It contains a reference to a “linked list” of Measure objects that belong to that voice by keeping a pointer to the first measure of the voice.

Each Measure is a container of Note instances, each Note being associated with a start time, represented by a Fraction, relative the beginning of the measure. Measures are structured like nodes in a linked list, with pointers to subsequent measures. They can have multiple pointers if the measure is the end of a repeat or part of a multiple ending sequence (see below)

Linked Representation of "M0 |: M1 | M2 | M3 :| M4 |: M5 | [1 M6 :|[2 M7 | M8 |]"

$$\begin{tikzcd}
 & & M2 \arrow[dr, "1"] & & & & M6 \arrow[dl, "1"] & \\
M0 \arrow[r, "1"] & M1 \arrow[ur, "1"] & & M3 \arrow[ll, bend left, "1"] \arrow[r, "2"] & M4 \arrow[r, "1"] & M5 \arrow[ur, "1"] \arrow[dr, "2"] & & \\
 & & & & & & M7 \arrow[r, "1"] & M8
\end{tikzcd}$$

Therefore, the same Measure object can represent both of its instantiations in a repeat.

Transformation into a Playable Sequence

Given a Piece object, we have a Visitor traverse through the Voices (e.g. the linked lists of Measures), and in accordance to the rules of transcription, transcribe MIDI-format playable music for the sequence player.

The Visitor will traverse each Voice in a given piece, and traverse the linked list of Measures in each voice. In transcribing the notes in a measure, Visitor will track global time in order to correctly time each note in the piece especially given that there may be multiple voicings and that Note objects in a measure only have descriptors of their relative times in each given measure.

MeasureIterator is an Iterator which helps the Visitor accurately navigate through the ’linked list of Measures’.

In the case of multiple endings, if the Visitor has seen a specific measure before (measured by reference NOT value, as two different measures may contain the same musical content), it will follow the second pointer to the next measure rather than the first pointer.

3. List of Classes and Data types

The following list shows all the classes that will be implemented for the ABC music player. They are grouped by functionality as shown by the boxes in the above dependency diagram. For the data type classes that describe the music

Lexer Classes

```
public class Lexer {
	private final List<TokenType> types;
	public Lexer(List<TokenType> types)
	//Lexes the passed string into Tokens, returns a list of tokens
	public List<Token> lex(String input) throws RuntimeException
}

public class Token {
	public final TokenType type;
	public final String contents;
	public Token(String contents, TokenType type)
	public String toString()
}

public class TokenType {
	public final String name;
	public final Pattern pattern;
	public TokenType(String name, Pattern pattern)
}

public class LexerTest {
	//Test file
}
```

Player Classes

Parser

```
public class Parser {
	//Parse the given string as a Piece.
	public static Piece parse( String abcContents )
}
```

Musical Data Type

MUTABLE

```
public class Piece {

	//The title of the piece.
	private String title;
	//The composer of the piece.
	private String composer;
	//Track Number of the piece.
	private int trackNumber;
	//Default length or duration of a note.
	private Fraction defaultNoteLength;
	//It determines the sum of the durations of all notes within a measure
	private Fraction meter;
	// The number of default-length notes per minute.
	private int tempo;
	//Determines the key signature for the piece.
	private String key;
	//The List of all the starting measures for each voice.
	private List<Voice> voices;
	//The (largest, ideally) smallest division needed such that the length of each
	note (and rest) is an integer multiple.
	private Fraction smallestDivision;

	// get the meter for the piece
	public Fraction getMeter()
	// set the meter for the piece
	public void setMeter(Fraction meter)
	// get the tempo for the piece
	public int getTempo()
	// set the tempo for the piece
	public void setTempo(int tempo)
	// get the smallest note division for the piece
	public Fraction getSmallestDivision()
	// set the smallest note division for the piece
	public void setSmallestDivision(Fraction smallestDivision)

	// Get starting measure of this voice
	public Measure getStart()
	// Get last measure of this voice
	public Measure getEnd()
	// Get a measure iterator, starting from this voice’s start measure
	public Iterator<Measure> iterator()
}
```

IMMUTABLE

```
public class Voice implements Iterable<Measure> {
        public final String name;
        private Measure firstMeasure;
        public Voice(String name, Measure firstMeasure)
        Measure start;
        //The measure we are currently concerned with.
        Measure current;
        //A Map containing how many times we have seen each measure by reference.
        Map<Measure, Integer> timesSeen;
        public MeasureIterator(Measure start)
        //Check if there’s a next measure.
        public boolean hasNext()
        //Return the next measure.
        public Measure next()
        //This method always throws an error - prevents removing measures.
        public void remove() throws RuntimeException
}
```

MUTABLE (contains methods to changing the measure’s pointers to other measures and adding or modifying what notes are in the method)

```
public class Measure implements Iterable<Measure> {
        //Length of the measure.
        private final Fraction length;
        //A list of notes, each associated with their start times
        private List<Pair<Note, Fraction>> notes;
        //The typical next measure in the larger piece.
        private Measure next;
        // An alternate next measure, e.g. the escape from a repeat.
        private Measure alternateNext = null;
        //Full constructor for Measure
        public Measure(Measure next, Measure alternateNext,List<Pair<Note, Fraction>>
        notes, Fraction length) throws NoteOutOfBoundsException
        //Constructor with a list of empty notes
        public Measure(Measure next, Measure alternateNext, Fraction length)
        //Constructs measure only based off the next Measure
        public Measure(Measure next, Fraction length)
        //Default constructor
        public Measure( Fraction length )
        //Returns an iterator which will start with this measure, and continue until the
        end of the piece is reached.
        public Iterator<Measure> iterator()
        //Getter for this.next
        public Measure getNext()
        //Setter for this.next
        public void setNext(Measure next)
        //Getter for this.alternateNext
        public Measure getAlternateNext()
        //Setter for this.alternateNext
        public void setAlternateNext(Measure alternateNext)
        //Getter for this.notes
        public List<Pair<Note, Fraction>> getNotes()
        // Safe method to add a note to this measure.
        public void addNote(Note note, Fraction startTime ) throws
        NoteOutOfBoundsException
}

IMMUTABLE
public class Note {
        public final Fraction duration;
        public final Pitch pitch;
        public Note(Fraction duration, Pitch pitch)
}

IMMUTABLE
public class CircleOfFifths {
        //return the key signature for the given key
        public getKeySignature(Note n)
}
```

Visitor

```
public class PieceVisitor {
        // Processes a Piece to play
        public static SequencePlayer process(Piece piece)
                        throws MidiUnavailableException, InvalidMidiDataException

        // Convert time (in fractional length) to ticks
        private static int fractionToTicks(Fraction time, Fraction divisionLength)

}

public class MeasureIterator implements Iterator<Measure> {
        //The measure we are currently initialized with.
        // get the key for the piece
        public String getKey()
        // set the key for the piece
        public void setKey(String key)
        // get the title for the piece
        public String getTitle()
        // set the title for the piece
        public void setTitle(String title)
        // get the composer name for the piece
        public String getComposer()
        // set the composer name for the piece
        public void setComposer(String composer)
        // get the track number for the piece
        public int getTrackNumber()
        // set the track number for the piece
        public void setTrackNumber(int trackNumber)
        // get the default note length for the piece
        public Fraction getDefaultNoteLength()
        // set the default note length for the piece
        public void setDefaultNoteLength(Fraction defaultNoteLength)
        // get the voices of this Piece
        public List<Voice> getVoices()
        // set the voices of this Piece
        public void setVoices(List<Voice> voices)

}
```

Utilities

```
//Represenation of a fraction, useful for calculating the duration of a note and ensuring
//that measures contain the correct length of notes
public class Fraction {
	public final int numerator;
	public final int denominator;
	public Fraction(int value)
	public Fraction(int numerator, int denominator)

	// Finds gcd of two integers
	public static int gcd(int first, int second)
	// Finds lcm of two integers
	public static int lcm(int first, int second)
	// Finds sum of this plus a Fraction
	public Fraction plus(Fraction other)
	// Finds sum of this plus an int
	public Fraction plus(int other)
	// Finds difference of this and a Fraction
	public Fraction minus(Fraction other)
	// Finds difference of this and an int
	public Fraction minus(int other)
	// Finds product of this and a Fraction
	public Fraction times(Fraction other)
	// Finds product of this and an int
	public Fraction times(int other)
	// Finds quotient of this and a Fraction
	public Fraction quotient(Fraction other)
	// Finds quotient of this and an int
	public Fraction quotient(int other)
	// Finds the inverse of this
	public Fraction inverse() throws FractionValueException
	// Finds float approximation of this
	public float approximation()
	// Finds hash of this
	public int hashCode()
	// Equals method for Fraction class
	public boolean equals(Object other)
	// Express this Fraction as a String
	public String toString()
}

// Custom exception for handling errors dealing with fractions
public class FractionValueException extends IllegalArgumentException {
	public FractionValueException(String message)
}

//Class representing a Pair of generic types
public class Pair<First, Second> {
	public First first;
	public Second second;
	public Pair(First first, Second second)
```

```
        // Finds hash of this
        public int hashCode()
        // Equals method for Pair objects.
        public boolean equals(Object other)
}
```

4. Testing

The modularity of our design allows that each major class can be tested. We would like to test the following classes for specific details as follows:

1. Lexer can separate every kind of token, throws exception of invalid tokens, and correctly handles null input.

2. Parser throws exceptions for all errors; correctly parses Measures with both basic notes and more complex tokens (chords, tuples); correctly creates nested repeat structures and multiple endings.

3. Visitor ensures global timekeeping is working and correctly translates to MIDI notes; correctly calculates a ’base’ tick/quarter note from the different beat divisions; correctly traverses through repeats and alternate endings; and ensures that errors thrown in Lexer/Parser also stop its actions.

In addition to testing individual modules, we must test the entire system as a whole using integration tests, to make sure that the components interface correctly with each other.