Testing Document

Tab2XML
Group 7

Joshua Genat
Andy Lin
Uthithmenon Ravitharan
Nicolae Semionov

Table of Contents

1.0 Introduction

1.1 Purpose

The purpose of this document is to describe all JUnit test cases, the classes they are designed to test, how they were derived and why they are useful.

2.0 Guitar Converter

2.1 Class description

The main function of this class is to accept a 2d character array parsed from a tab, and to convert into a form that is more accessible to the xml converters.

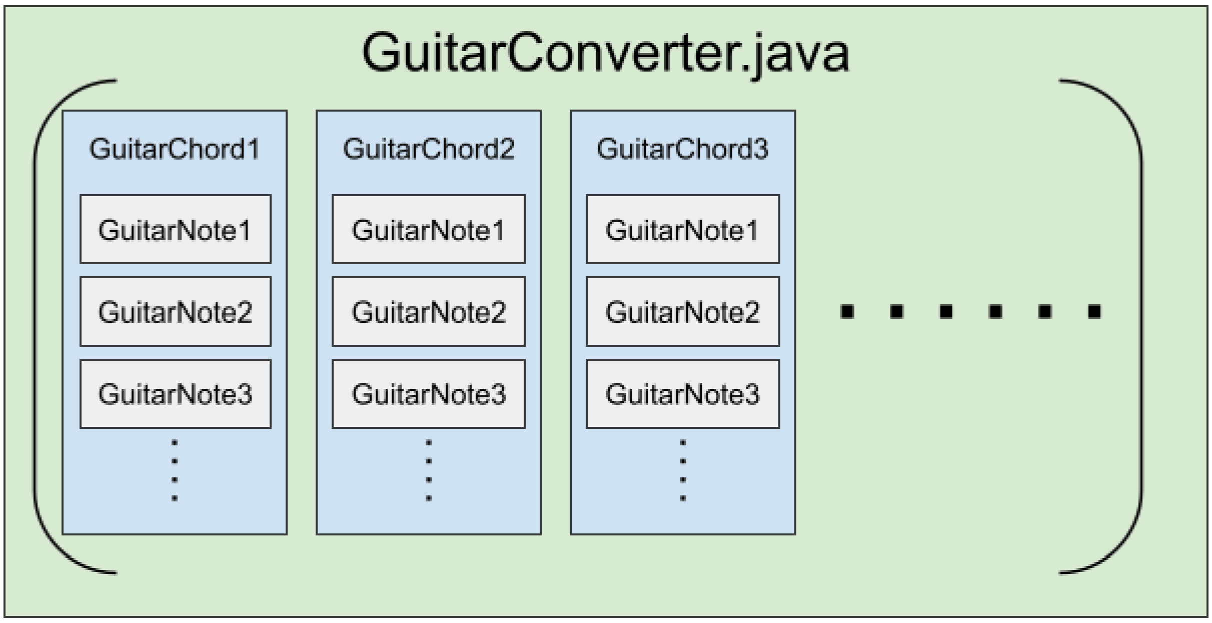
Other classes used by GuitarConverter.java

- GuitarChord.java: This class is used to create objects of guitar chords. GuitarChords store an array of GuitarNotes, and include methods to manipulate multiple notes in an organized fashion
- GuitarNotes: Objects used by the program's xml converters. Their main purpose it to have a convenient way of accessing note properties at a particular position in a tab such as;
    - String note
    - String type
    - int alter
    - int octave
    - int string
    - int fret
    - int duration
    - int voice
    - char step
    - boolean isChord
- GuitarNotes also need a way to represent hammer-ons and pull-offs, this is done by using booleans, and pointers to the hammer-on or

pull-off note. For grace notes there is just a simple boolean, no pointers needed

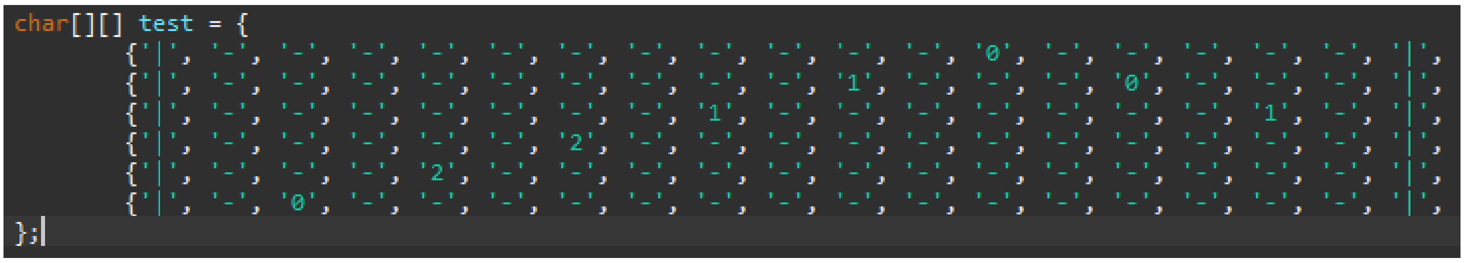
- Boolean isHammer
- GuitarNotes hammerTo
- Boolean isPull
- GuitarNotes pullTo
- Boolean isGrace

The class returns an array of GuitarChord objects. A GuitarChord is simply a convenient way of organizing arrays of GuitarNotes with extra functionality.



2.2 JUnit Test

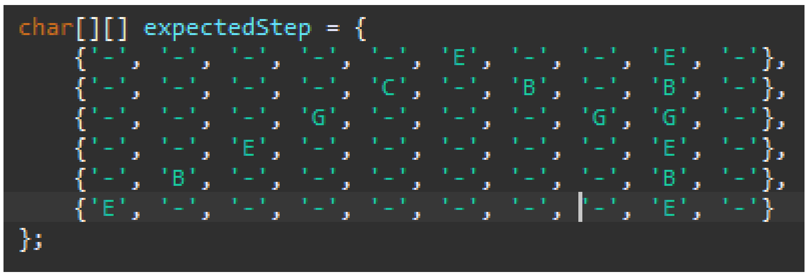
The JUnit test GuitarConverterTester.java aims to ensure all note properties are getting assigned proper values. The tester passes a sample input provided in the course wiki as a 2d array of characters.



```
char[][] test = {
        {'|', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '0', '-', '-', '-', '-', '-', '|',
        {'|', '-', '-', '-', '-', '-', '-', '-', '-', '-', '1', '-', '-', '-', '0', '-', '-', '-', '|',
        {'|', '-', '-', '-', '-', '-', '-', '-', '1', '-', '-', '-', '-', '-', '-', '-', '1', '-', '|',
        {'|', '-', '-', '-', '-', '-', '2', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '|',
        {'|', '-', '-', '-', '2', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '|',
        {'|', '-', '0', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '|',
};
```

The resulting output of GuitarConverter.converter(test) is saved in an array of GuitarChords named ‘result’.

The result is then compared using a randomly generated index with an array of expected outputs. For example, here is a 2d array of expected step chars.



```
char[][] expectedStep = {
    {'-', '-', '-', '-', '-', 'E', '-', '-', 'E', '-'},
    {'-', '-', '-', '-', 'C', '-', 'B', '-', 'B', '-'},
    {'-', '-', '-', 'G', '-', '-', '-', 'G', 'G', '-'},
    {'-', '-', 'E', '-', '-', '-', '-', '-', 'E', '-'},
    {'-', 'B', '-', '-', '-', '-', '-', '-', 'B', '-'},
    {'E', '-', '-', '-', '-', '-', '-', '-', 'E', '-'}
};
```

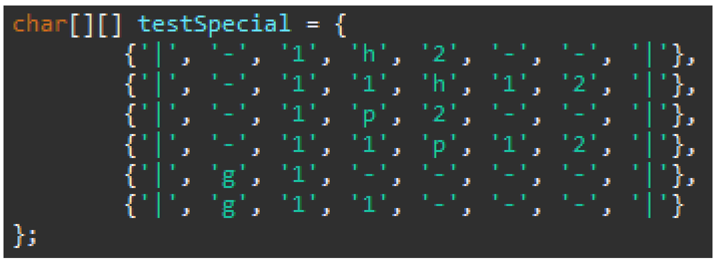
This 2d char step array is used to compare with the steps in GuitarChords array ‘result’ in the following lines of code

```
@Test
void stepTest() {
      assertEquals(result[i[0]].notes[i[1]].step,expectedStep[i[1]][i[0]]);
}
```

This process is repeated for all note properties

A different approach had to be used for special notes. Since the destination of a hammer-on note does not appear in the list of notes, it cannot be easily indexed. A manual approach had to be taken in order to ensure the method worked correctly. This is true for pull-offs as well.

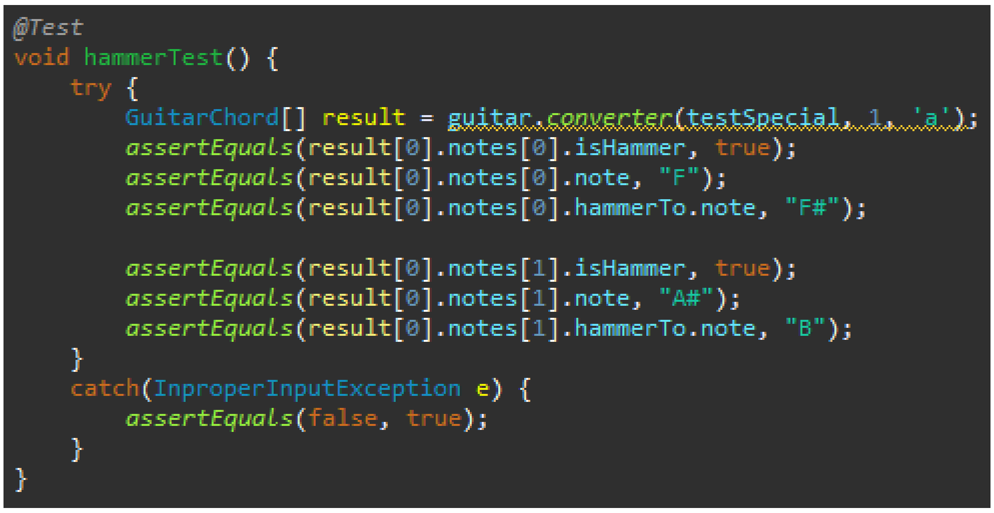
This is the test input used to test a series of hammer-ons, pull-offs and grace notes.



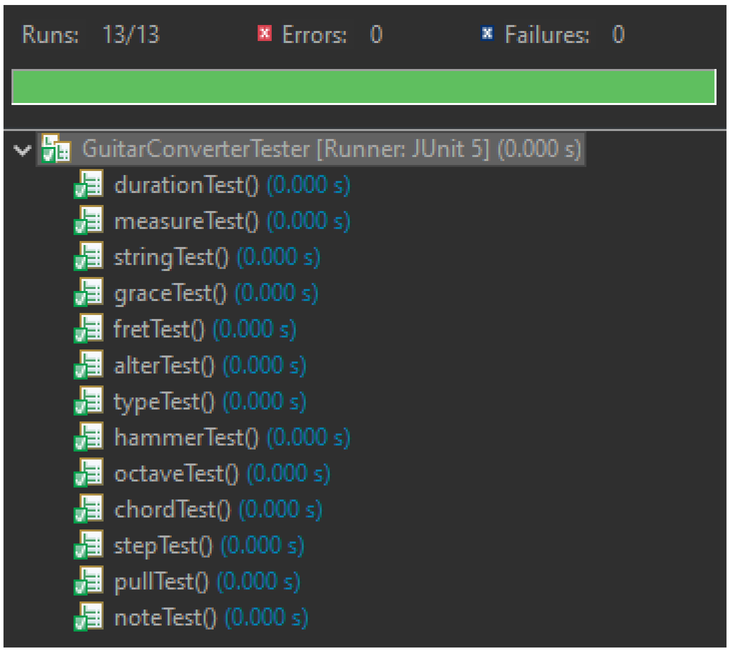
```
char[][] testSpecial = {
        {'|', '-', '1', 'h', '2', '-', '-', '|'},
        {'|', '-', '1', '1', 'h', '1', '2', '|'},
        {'|', '-', '1', 'p', '2', '-', '-', '|'},
        {'|', '-', '1', '1', 'p', '1', '2', '|'},
        {'|', 'g', '1', '-', '-', '-', '-', '|'},
        {'|', 'g', '1', '1', '-', '-', '-', '|'}
};
```

```
|-1h2--|
|-11h12|
|-1p2--|
|-11p12|
|g1----|
|g11---|
```

The result is manually compared to the expected output

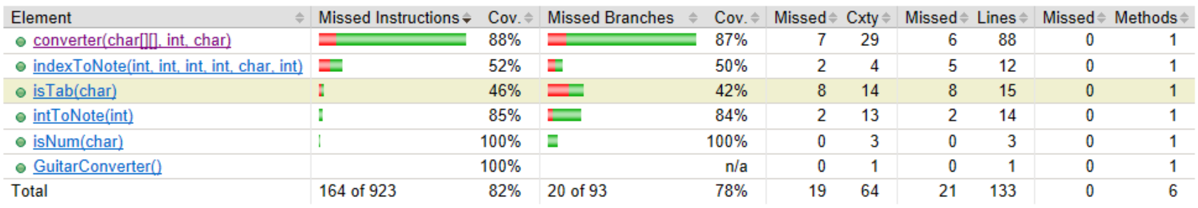

```
@Test
void hammerTest() {
    try {
        GuitarChord[] result = guitar.converter(testSpecial, 1, 'a');
        assertEquals(result[0].notes[0].isHammer, true);
        assertEquals(result[0].notes[0].note, "F");
        assertEquals(result[0].notes[0].hammerTo.note, "F#");

        assertEquals(result[0].notes[1].isHammer, true);
        assertEquals(result[0].notes[1].note, "A#");
        assertEquals(result[0].notes[1].hammerTo.note, "B");
    }
    catch(InproperInputException e) {
        assertEquals(false, true);
    }
}
```

These tests are sufficient since the array index is randomly generated. A change that results in the incorrect assignment of a note property will likely be spotted by the tester on the first run through. The tests also cover most of the vital methods located in GuitarNotes, GuitarChords and GuitarConverter.


```
Runs:  13/13        Errors:  0        Failures:  0

GuitarConverterTester [Runner: JUnit 5] (0.000 s)
    durationTest() (0.000 s)
    measureTest() (0.000 s)
    stringTest() (0.000 s)
    graceTest() (0.000 s)
    fretTest() (0.000 s)
    alterTest() (0.000 s)
    typeTest() (0.000 s)
    hammerTest() (0.000 s)
    octaveTest() (0.000 s)
    chordTest() (0.000 s)
    stepTest() (0.000 s)
    pullTest() (0.000 s)
    noteTest() (0.000 s)
```

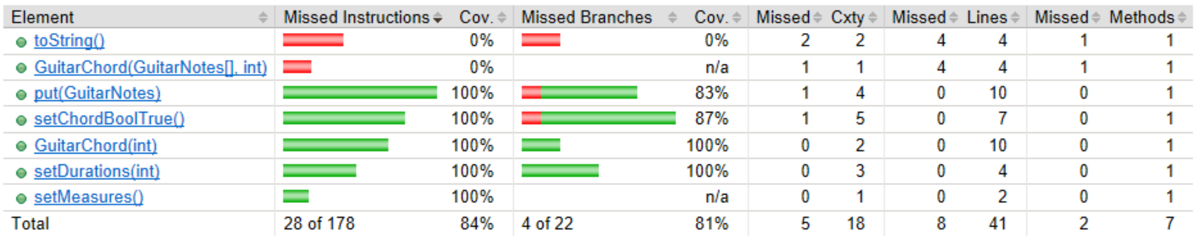
Test coverage is very high for all vital methods

GuitarConverter



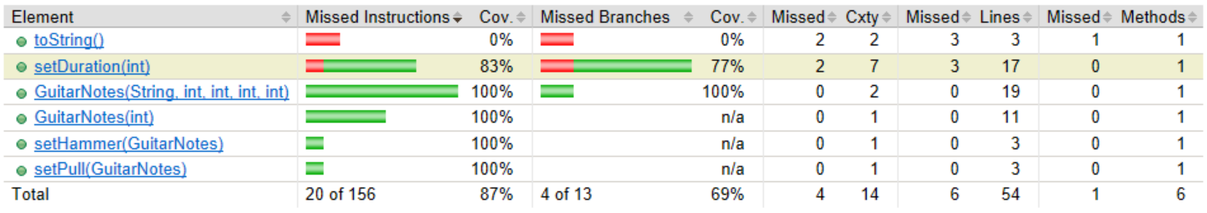
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
converter(char[][], int, char)	🟥🟩	88%	🟥🟩	87%	7	29	6	88	0	1
indexToNote(int, int, int, int, char, int)	🟥🟩	52%	🟥🟩	50%	2	4	5	12	0	1
isTab(char)	🟥🟩	46%	🟥🟩	42%	8	14	8	15	0	1
intToNote(int)	🟩	85%	🟥🟩	84%	2	13	2	14	0	1
isNum(char)	🟩	100%	🟩	100%	0	3	0	3	0	1
GuitarConverter()		100%		n/a	0	1	0	1	0	1
Total	164 of 923	82%	20 of 93	78%	19	64	21	133	0	6

GuitarChord



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
toString()	🟥	0%	🟥	0%	2	2	4	4	1	1
GuitarChord(GuitarNotes[], int)	🟥	0%		n/a	1	1	4	4	1	1
put(GuitarNotes)	🟩	100%	🟥🟩	83%	1	4	0	10	0	1
setChordBoolTrue()	🟩	100%	🟥🟩	87%	1	5	0	7	0	1
GuitarChord(int)	🟩	100%	🟩	100%	0	2	0	10	0	1
setDurations(int)	🟩	100%	🟩	100%	0	3	0	4	0	1
setMeasures()	🟩	100%		n/a	0	1	0	2	0	1
Total	28 of 178	84%	4 of 22	81%	5	18	8	41	2	7

GuitarNotes



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
toString()	🟥	0%	🟥	0%	2	2	3	3	1	1
setDuration(int)	🟥🟩	83%	🟥🟩	77%	2	7	3	17	0	1
GuitarNotes(String, int, int, int, int)	🟩	100%	🟩	100%	0	2	0	19	0	1
GuitarNotes(int)	🟩	100%		n/a	0	1	0	11	0	1
setHammer(GuitarNotes)	🟩	100%		n/a	0	1	0	3	0	1
setPull(GuitarNotes)	🟩	100%		n/a	0	1	0	3	0	1
Total	20 of 156	87%	4 of 13	69%	4	14	6	54	1	6

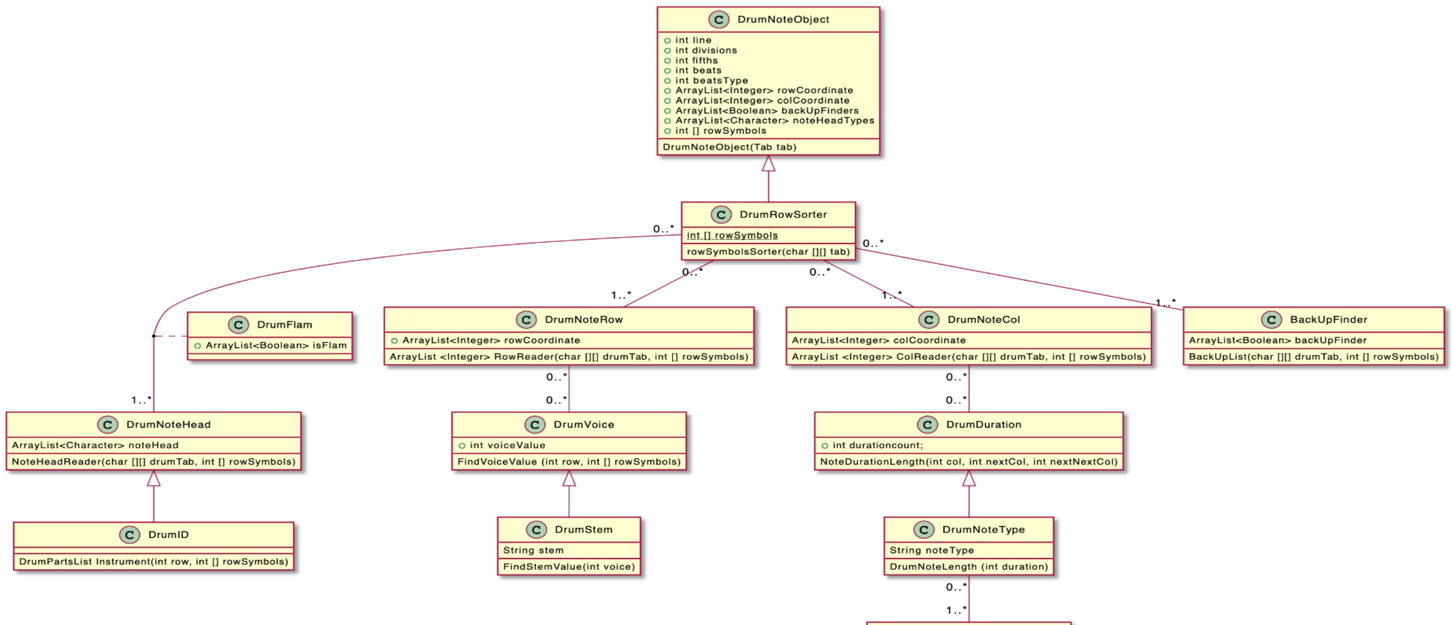
3.0 Drum Tester

3.1 Class description

Converting a drum tablature to MusicXML code requires the retrieval of many attributes from the tablature, such as:

- Coordinate of Notes and order in which they are played (row and column values)
- Duration of how long Notes are played (duration)
- Display octave and display Step
- Type of notes being played (Note Type)
- Voice and stem direction
- If note played is a chord
- If note played is part of a beam (Beam Number)

The main function of the Drum Tester class is to test the methods which are obtaining these attributes for the MusicXML convertor. The structural diagram of the Drum Convertor, shows that the four main classes which enable the rest of the classes to run are DrumNoteHead, DrumNoteRow, DrumNoteCol and BackUpFinder. This class will provide a test for each one of these classes, because these classes are the backbone of the drum converter, and make up upwards of eighty percent of its programming.



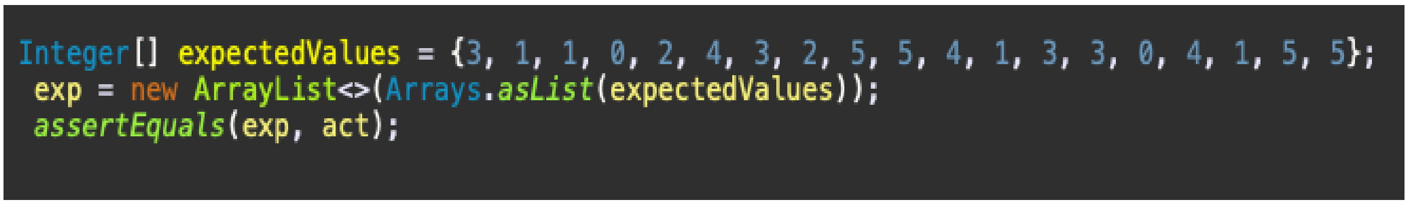
DrumNoteObject
int line
int divisions
int fifths
int beats
int beatsType
ArrayList<Integer> rowCoordinate
ArrayList<Integer> colCoordinate
ArrayList<Boolean> backUpFinders
ArrayList<Character> noteHeadTypes
int [] rowSymbols
DrumNoteObject(Tab tab)
DrumRowSorter
int [] rowSymbols
rowSymbolsSorter(char [][] tab)
0..*
1..*
DrumFlam
ArrayList<Boolean> isFlam
DrumNoteRow
ArrayList<Integer> rowCoordinate
ArrayList <Integer> RowReader(char [][] drumTab, int [] rowSymbols)
DrumNoteCol
ArrayList<Integer> colCoordinate
ArrayList <Integer> ColReader(char [][] drumTab, int [] rowSymbols)
BackUpFinder
ArrayList<Boolean> backUpFinder
BackUpList(char [][] drumTab, int [] rowSymbols)
DrumNoteHead
ArrayList<Character> noteHead
NoteHeadReader(char [][] drumTab, int [] rowSymbols)
DrumVoice
int voiceValue
FindVoiceValue (int row, int [] rowSymbols)
DrumDuration
int durationcount;
NoteDurationLength(int col, int nextCol, int nextNextCol)
DrumID
DrumPartsList Instrument(int row, int [] rowSymbols)
DrumStem
String stem
FindStemValue(int voice)
DrumNoteType
String noteType
DrumNoteLength (int duration)

Test 1 - RowArrayListTest()

The test case takes a 2d character array (input from user) and returns the row values at which a percussion instrument is played in the correct order they would appear in MusicXML code. In a percussion MusicXML file, voice one notes are listed first, followed by voice two notes. The following test should fulfill this requirement (voice number depends on the type of percussion instrument being played.

This tester class is very important for the conversion of tabulate to Music XML, because knowing the correct order at which notes are played, is instrumental for creating accurate sheet music. Knowing the row at which a note is played is essential for assigning a voice value and stem direction for each note.

Expected ArrayList Output:



```
Integer[] expectedValues = {3, 1, 1, 0, 2, 4, 3, 2, 5, 5, 4, 1, 3, 3, 0, 4, 1, 5, 5};
 exp = new ArrayList<>(Arrays.asList(expectedValues));
 assertEquals(exp, act);
```

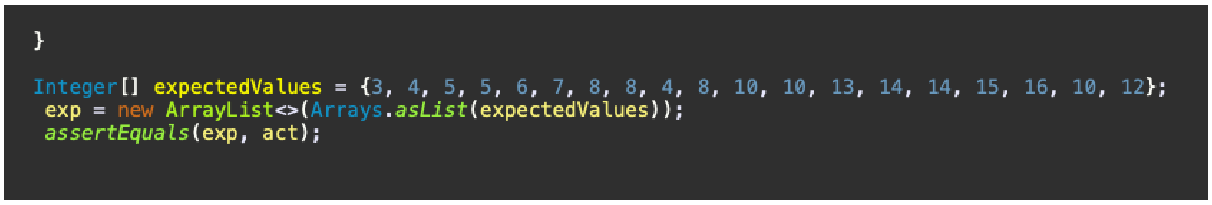
Sufficiency: This test is sufficient as the output combined with the ColArrayListTest() list the correct order at which the drum notes should be listed in a MusicXML file. Which is a great starting point for converting a drum tabulate to MusicXML.

Test 2 - ColArrayListTest()

This test takes in a 2d character array, and returns an Arraylist of the integers. These integers are the column values of when the percussion notes are played and they are listed in the correct order for an MusicXML file. For example if notes are played at row 4, column 1, the integer 1 would be in the Arraylist.

This test is very important because the column values are used to calculate duration, check if a note is a chord, and to find note type. Furthermore, the column values paired with duration are used to see if the tablature entered is invalid, for example a tablature with a duration of 17 is not supported by our system and will send an error.

Expected Arraylist Output:



```
}

Integer[] expectedValues = {3, 4, 5, 5, 6, 7, 8, 8, 4, 8, 10, 10, 13, 14, 14, 15, 16, 10, 12};
 exp = new ArrayList<>(Arrays.asList(expectedValues));
 assertEquals(exp, act);
```

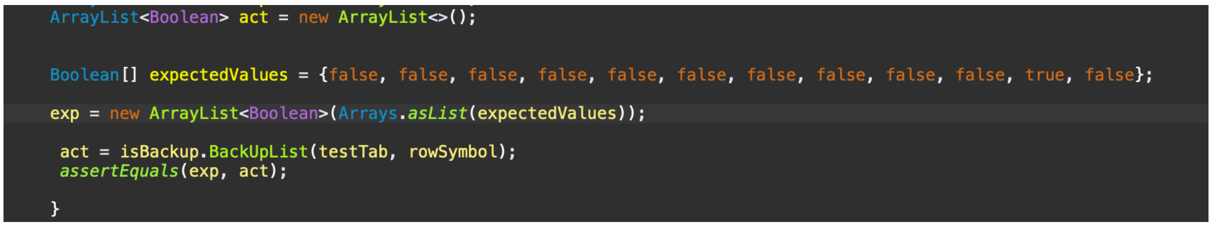
Sufficiency: This test is sufficient as the output combined with the RowArrayListTest() list the correct order at which the drum notes should be listed in a MusicXML file. Which is a great starting point for converting a drum tabulate to MusicXML.

Test 3 - BackUpTest()

This test takes in a 2d character array, and returns an Arraylist of the boolean values. These boolean values tell the user whether or not to paste the <Back up> tag. The back-up tag should be printed in the MusicXML code, when the notes listed goes from voice one to voice two.

This test is very important because the back-up tags and measure number tags alternate. Furthermore, the back-up tag is important for the accuracy of the music, when putting the MusicXML code into a Musicxml reader such as soundslice.

Expected Arraylist Output:



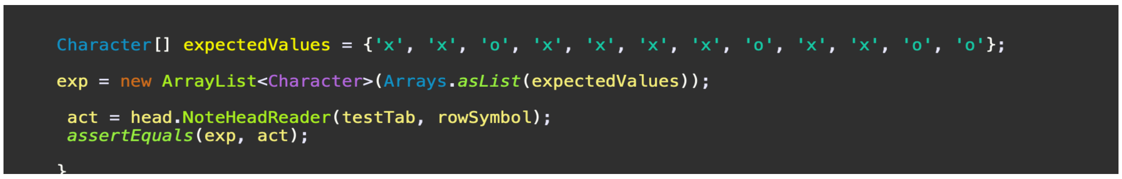
```
ArrayList<Boolean> act = new ArrayList<>();

Boolean[] expectedValues = {false, false, false, false, false, false, false, false, false, false, true, false};

exp = new ArrayList<Boolean>(Arrays.asList(expectedValues));

 act = isBackup.BackUpList(testTab, rowSymbol);
 assertEquals(exp, act);

}
```

Sufficiency: This test is sufficient as the placement of the back up tag is very important for making an accurate MusicXML code. An incorrect backup value will ruin the music sheet produced when using a MusicXML reader.

Test 4 - NoteHeadTest()

This test takes in a 2d character array, and returns an Arraylist of the character values. These character values tell the system which drum technique is being used to play the instrument. For example, ‘o’ means strike and is usually used for drums, while ‘x’ means strike cymbal. This class is needed to identify whether an open or closed Hi-Hat is being played. Which in turn is needed to correctly identify the ID number for every note in the tablature.

Expected Arraylist Output:

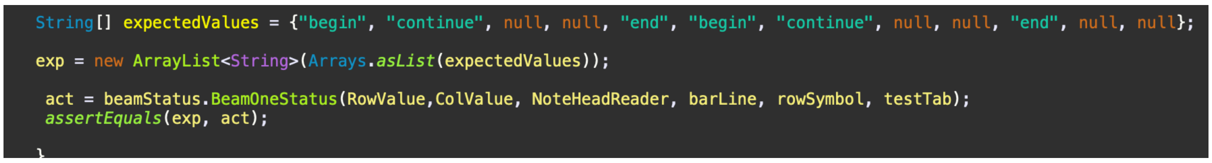


```
Character[] expectedValues = {'x', 'x', 'o', 'x', 'x', 'x', 'x', 'o', 'x', 'x', 'o', 'o'};

exp = new ArrayList<Character>(Arrays.asList(expectedValues));

 act = head.NoteHeadReader(testTab, rowSymbol);
 assertEquals(exp, act);
```

Test 5 & 6 - BeamOneStatusTest and BeamTwoStatusTest()

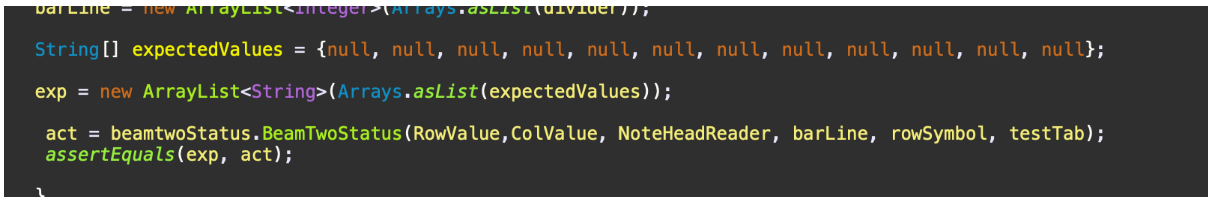
Beam Numbers were the hardest attributes to find for drum conversion. To give context, both of the beam number tests require six parameters each. BeamOneStatusTest() finds out if the note in question is part of a beam and if so, returns a string of the position the note has in a beam; such as ‘start’, ‘continue’ and ‘end’. BeamTwoStatusTest() does the same thing except for Beam Number 2: our system only supports up to 16th notes, so only two beams are possible.

Expected Arraylist Output of BeamOneStatusTest:



```
String[] expectedValues = {"begin", "continue", null, null, "end", "begin", "continue", null, null, "end", null, null};

exp = new ArrayList<String>(Arrays.asList(expectedValues));

 act = beamStatus.BeamOneStatus(RowValue,ColValue, NoteHeadReader, barLine, rowSymbol, testTab);
 assertEquals(exp, act);

}
```

Expected Arraylist Output of BeamTwoStatusTest:



```
barLine = new ArrayList<Integer>(Arrays.asList(divider));

String[] expectedValues = {null, null, null, null, null, null, null, null, null, null, null, null};

exp = new ArrayList<String>(Arrays.asList(expectedValues));

 act = beamtwoStatus.BeamTwoStatus(RowValue,ColValue, NoteHeadReader, barLine, rowSymbol, testTab);
 assertEquals(exp, act);

}
```

Sufficiency: These tests are sufficient because beam numbers are what separate a messy/amatar music sheet from a professional looking one. Also as you can see from our coverage page listed at the end, the beam number class was the biggest in the program by a significant amount. Therefore two test cases was the only option to make sure everything in the class was covered.

4.0 Parser Tester

4.1 Class description

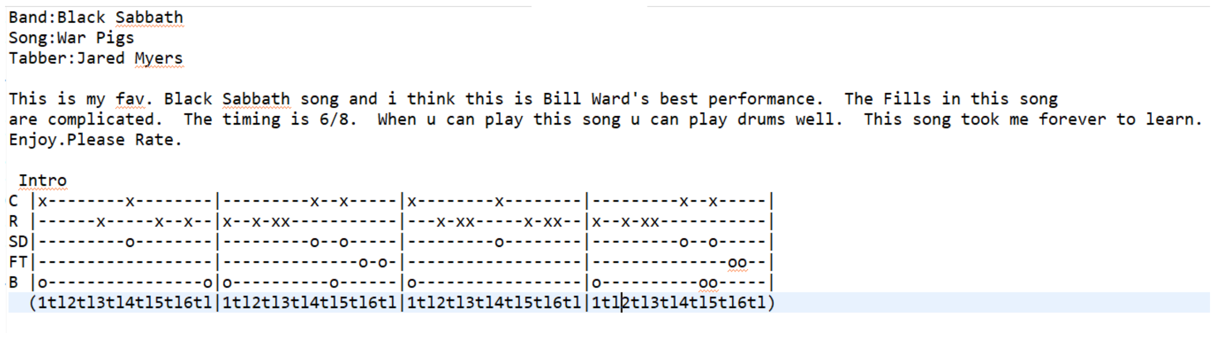
The main function of these classes is to take in a file or text and extract the necessary information.

Test 1 - ParserRemoveTrashTest()

This test run checks to see if the parser removes excess strings such as lyrics and symbol definers and gets rid of it. This will allow us to extract only the tablature parts of the file and make it easier for conversion.

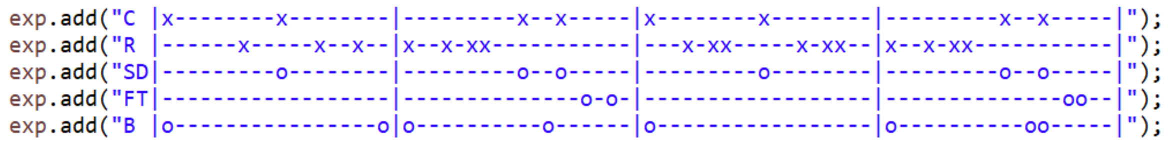
Implementation: Uploaded a sample file and ran it through the parser, I took the ArrayList<String> that the parser makes and I compared it to a ArrayList<String> that I manually made that just had the tablature and assert equal.

Input:



```
Band:Black Sabbath
Song:War Pigs
Tabber:Jared Myers

This is my fav. Black Sabbath song and i think this is Bill Ward's best performance.  The Fills in this song
are complicated.  The timing is 6/8.  When u can play this song u can play drums well.  This song took me forever to learn.
Enjoy.Please Rate.

 Intro
C |x--------x--------|---------x--x-----|x--------x--------|---------x--x-----|
R |------x-----x--x--|x--x-xx-----------|---x-xx-----x-xx--|x--x-xx-----------|
SD|---------o--------|---------o--o-----|---------o--------|---------o--o-----|
FT|------------------|--------------o-o-|------------------|--------------oo--|
B |o----------------o|o---------o-------|o-----------------|o---------oo------|
  (1tl2tl3tl4tl5tl6tl|1tl2tl3tl4tl5tl6tl|1tl2tl3tl4tl5tl6tl|1tl2tl3tl4tl5tl6tl)
```

Expected:



```
exp.add("C |x--------x--------|---------x--x-----|x--------x--------|---------x--x-----|");
exp.add("R |------x-----x--x--|x--x-xx-----------|---x-xx-----x-xx--|x--x-xx-----------|");
exp.add("SD|---------o--------|---------o--o-----|---------o--------|---------o--o-----|");
exp.add("FT|------------------|--------------o-o-|------------------|--------------oo--|");
exp.add("B |o----------------o|o---------o-------|o-----------------|o---------oo------|");
```

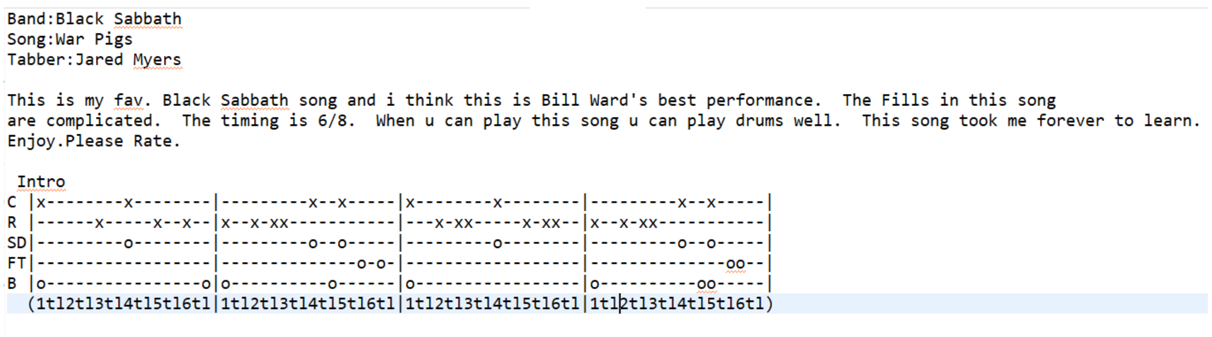
Sufficiency: This test is Sufficient because the program has shown that it can extract the tablature from numbers, symbols, and strings.

Test 2 - isDrumTypeCorrect()

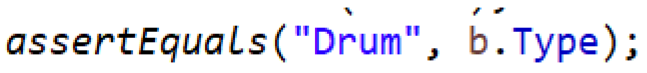
This test run checks to see if the parser can determine the tablature type and in this case if it can determine that the tablature is a drum type

Implementation: Uploaded a sample file of a drum tablature and ran it through the parser. I took the String type that the parser generates and compare it to “Drum”

Input:



```
Band:Black Sabbath
Song:War Pigs
Tabber:Jared Myers

This is my fav. Black Sabbath song and i think this is Bill Ward's best performance.  The Fills in this song
are complicated.  The timing is 6/8.  When u can play this song u can play drums well.  This song took me forever to learn.
Enjoy.Please Rate.

 Intro
C |x--------x--------|---------x--x-----|x--------x--------|---------x--x-----|
R |------x-----x--x--|x--x-xx-----------|---x-xx-----x-xx--|x--x-xx-----------|
SD|---------o--------|---------o--o-----|---------o--------|---------o--o-----|
FT|------------------|--------------o-o-|------------------|--------------oo--|
B |o----------------o|o---------o-------|o-----------------|o---------oo------|
  (1tl2tl3tl4tl5tl6tl|1tl2tl3tl4tl5tl6tl|1tl2tl3tl4tl5tl6tl|1tl2tl3tl4tl5tl6tl)
```

Expected: 

```
assertEquals("Drum", b.Type);
```

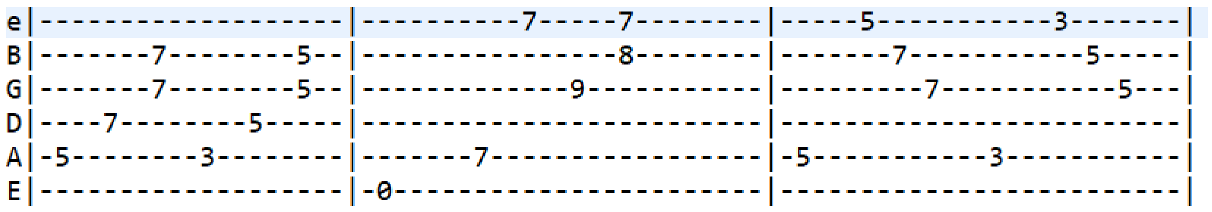
Sufficiency: This test is sufficient as this shows us that the parser can determine the type of tablature and combined with the other 3 will show that it covers all bases.

Test 3 - isGuitarTypeCorrect()

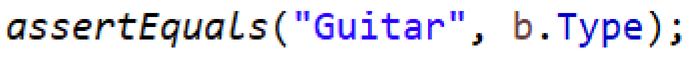
This test run checks to see if the parser can determine the tablature type and in this case if it can determine that the tablature is a guitar

Implementation: Uploaded a sample file of a guitar tablature and ran it through the parser. I took the String type that the parser generates and compare it to “Guitar”

Input:



```
e|-------------------|----------7-----7--------|-----5-----------3-------|
B|-------7--------5--|----------------8--------|-------7-----------5-----|
G|-------7--------5--|-------------9-----------|---------7-----------5---|
D|----7--------5-----|-------------------------|-------------------------|
A|-5--------3--------|-------7-----------------|-5-----------3-----------|
E|-------------------|-0-----------------------|-------------------------|
```

Expected: 

```
assertEquals("Guitar", b.Type);
```

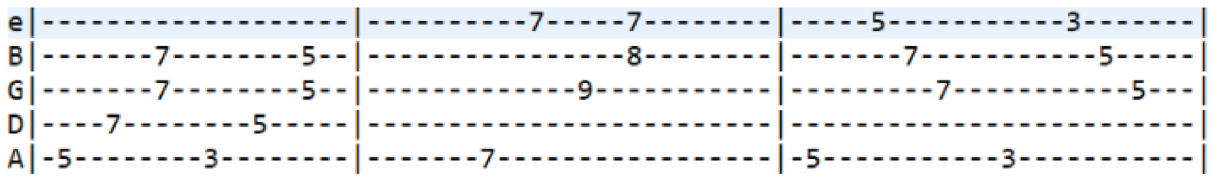
Sufficiency: This test is sufficient as this shows us that the parser can determine the type of tablature and combined with the other 3 will show that it covers all bases.

Test 4 - isBassTypeCorrect()

This test run checks to see if the parser can determine the tablature type and in this case if it can determine that the tablature is a Bass

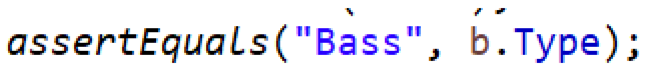
Implementation: Uploaded a sample file of a basstablature and ran it through the parser. I took the String type that the parser generates and compare it to “Bass”

Input:



```
e|-------------------|-------------------------|-------------------------|
B|-------7--------5--|----------------8--------|-------7-----------5-----|
G|-------7--------5--|------------9------------|---------7-----------5---|
D|----7--------5-----|-------------------------|-------------------------|
A|-5--------3--------|-------7-----------------|-5-----------3-----------|
```

Expected:



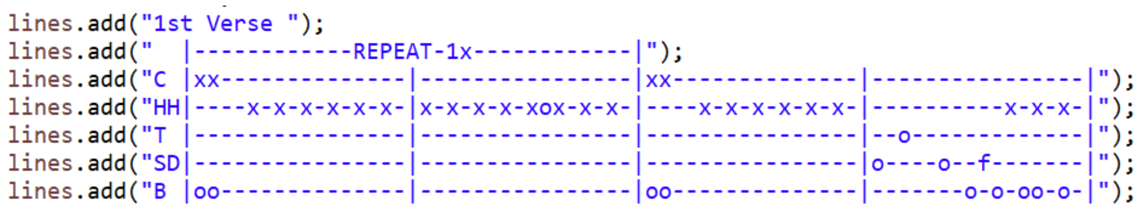
```
assertEquals("Bass", b.Type);
```

Sufficiency: This test is sufficient as this shows us that the parser can determine the type of tablature and combined with the other 3 will show that it covers all bases.

Test 5 - repeatType1()
This test run checks to see if the parser can interpret the repeat of kind |----repeatxN---|

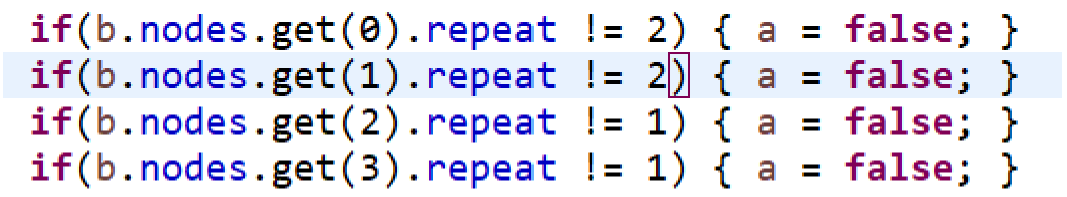
Implementation: Uploaded a sample file of a drum tablature that has a repeat of type |----repeatxN---|, and manually make the repeat numbers that are expected for each measure and compare it to each measure the tab class has made, and if they all equal assertEqual will pass.

Input:



```
lines.add("1st Verse ");
lines.add("  |------------REPEAT-1x------------|");
lines.add("C |xx--------------|----------------|xx--------------|----------------|");
lines.add("HH|----x-x-x-x-x-x-|x-x-x-x-xox-x-x-|----x-x-x-x-x-x-|----------x-x-x-|");
lines.add("T |----------------|----------------|----------------|--o-------------|");
lines.add("SD|----------------|----------------|----------------|o---o--f--------|");
lines.add("B |oo--------------|----------------|oo--------------|-------o-o-oo-o-|");
```

Expected:



```
if(b.nodes.get(0).repeat != 2) { a = false; }
if(b.nodes.get(1).repeat != 2) { a = false; }
if(b.nodes.get(2).repeat != 1) { a = false; }
if(b.nodes.get(3).repeat != 1) { a = false; }
```

Time of played will be 2, 2, 1, 1

Sufficiency: This test is sufficient as this shows us that the parser interprets the repeat and also finds the section that is required to be repeated. And extend it accordingly.

Test 6 - repeatType2()
This test run checks to see if the parser can interpret the repeat of kind xN where N is the number to be repeated

Implementation: Uploaded a sample file of a drum tablature that has a repeat of type xN, and manually make the repeat numbers that are expected for each measure and compare it to each measure the tab class has made, and if they all equal assertEqual will pass.

Input:

```
lines.add("C |xx--------------|----------------|xx--------------|----------------|");
lines.add("HH|----x-x-x-x-x-x-|x-x-x-x-xox-x-x-|----x-x-x-x-x-x-|----------x-x-x-|");
lines.add("T |----------------|----------------|----------------|--o-------------| x2");
lines.add("SD|----------------|----------------|----------------|o---o--f--------|");
lines.add("B |oo--------------|----------------|oo--------------|-------o-o-o-o-|");
```

Expected:

```
if(b.nodes.get(0).repeat != 2) { a = false; }
if(b.nodes.get(1).repeat != 2) { a = false; }
if(b.nodes.get(2).repeat != 2) { a = false; }
if(b.nodes.get(3).repeat != 2) { a = false; }
```

Checks to see if each measure is being played twice

Sufficiency: This test is sufficient as this shows us that the parser interprets the repeat. And adds it multiple times according to the number.

Test 7 - repeatType3()

This test run checks to see if the parser can interpret the repeat of kind ||**N| where N is the number to be repeated

Implementation: Uploaded a sample file of a Guitar tablature that has a repeat of type ||**N|, manually make the repeat numbers that are expected for each measure and compare it to each measure the tab class has made, and if they all equal assertEqual will pass.

Input:

```
lines.add("|-----------0-----||----------0--------4|");
lines.add("|---------0---0---||----------0--------||");
lines.add("|-------1-------1-||*---------1-------*||");
lines.add("|-----2-----------||*---------2-------*||");
lines.add("|---2-------------||------2---2--------||");
lines.add("|-0---------------||--0-------0--------||");
```

Expected:

```
if(b.nodes.get(0).repeat != 1) { a = false; }
if(b.nodes.get(1).repeat != 4) { a = false; }
```

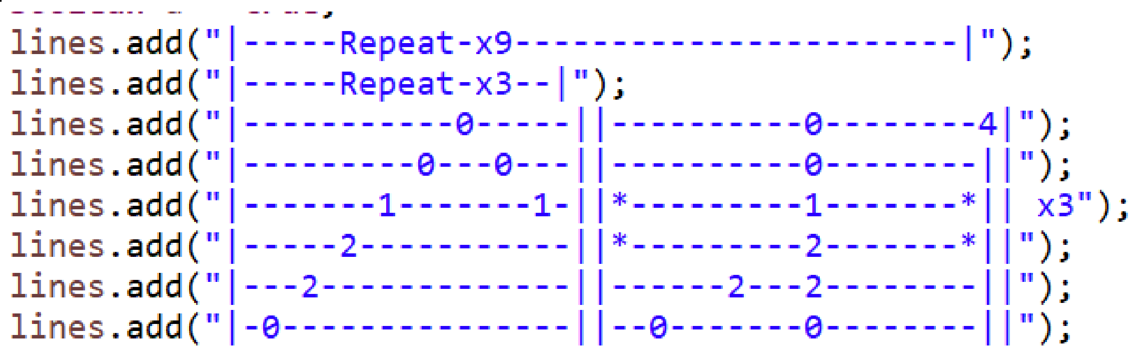
Measure 1 should be played once and measure 2 should be played 4 times.

Sufficiency: This test is sufficient as this shows us that the parser interprets the repeat.

Test 8 - repeatTypeAll()
This test run checks to see if the parser can interpret a mix of all 3 repeats, this will make the overall testing for repeats super sufficient as it tests all 3 at once.

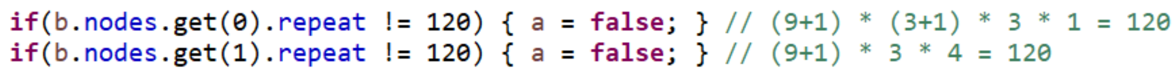
Implementation: uploaded a sample of a guitar tablature that contains all 3 repeats, and then manually make the repeat numbers that are expected for each measure and compare it to each measure the tab class has made, and if they all equal assertEqual will pass.

Input:



```
lines.add("|-----Repeat-x9-----------------------|");
lines.add("|-----Repeat-x3--|");
lines.add("|-----------0-----||----------0--------4|");
lines.add("|---------0---0---||----------0--------||");
lines.add("|-------1-------1-||*---------1-------*|| x3");
lines.add("|-----2-----------||*---------2-------*||");
lines.add("|---2-------------||------2---2--------||");
lines.add("|-0---------------||--0-------0--------||");
```

Expected:



```
if(b.nodes.get(0).repeat != 120) { a = false; } // (9+1) * (3+1) * 3 * 1 = 120
if(b.nodes.get(1).repeat != 120) { a = false; } // (9+1) * 3 * 4 = 120
```

Both measures should be played 120 times.

Sufficiency: This test is sufficient as this shows us that the parser interprets the all repeats, and shows that it can recognize all 3 at the same time without interfering with each other.

5.0 GUI Tester

5.1 Class description

The GUI class that contains all the components for the user to see.

Test 1 - convertBtn()
This test run to check if the converBtn exists at the start of making the GUI

Implementation: Uses FX Robot to startup the stage, and then verifying that the button is there.

Input: N/A

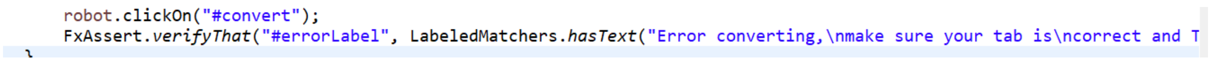
Expected: Button to be there

Sufficiency: This test is Sufficient because the program will know that the button is there, and another other button is initialized with it, so this acts as a test for all initial buttons

Test 2 - convertBtnClick()

This test run to check if the clicking the convertBtn without anything added will give out a warning to user.

Implementation: Uses FX Robot to startup the stage, and then using the robot to click the convert button without inputting a tab and then verifying the label is there.



```
    robot.clickOn("#convert");
    FxAssert.verifyThat("#errorLabel", LabeledMatchers.hasText("Error converting,\nmake sure your tab is\ncorrect and T
}
```

Input: N/A

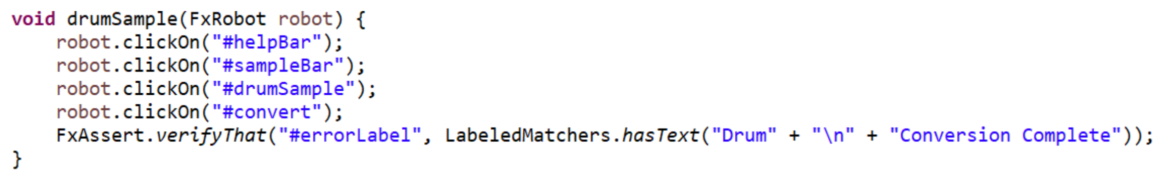
Expected: FX robot verification accepted.

Sufficiency: This test is Sufficient because the program will know that the error label exist, and this error label is reused for other warnings so it will verify a lot of the other warnings as well.

Test 3 - drumSample()

This test run to check if the clicking the drumSample button of the GUI is working

Implementation: Uses FX Robot to startup the stage, and then using the robot to to navigate to the drumSample button and click, then it will convert and verify that the drum tab was converted



```
void drumSample(FxRobot robot) {
    robot.clickOn("#helpBar");
    robot.clickOn("#sampleBar");
    robot.clickOn("#drumSample");
    robot.clickOn("#convert");
    FxAssert.verifyThat("#errorLabel", LabeledMatchers.hasText("Drum" + "\n" + "Conversion Complete"));
}
```

Input: N/A

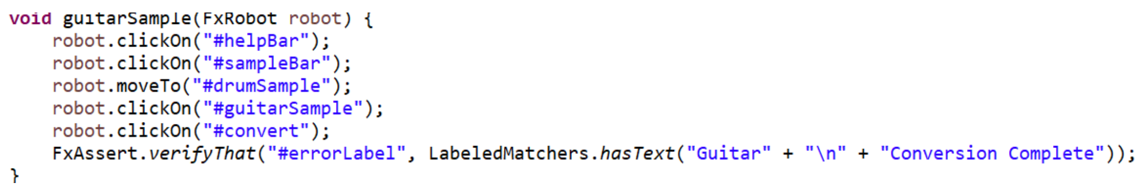
Expected: FX robot verification accepted.

Sufficiency: This test is Sufficient because the program will know there is a drum sample that users can work off of, this will also verify that drum conversions of the sample tabs also works, as it verifies both using the success label.

Test 4 - guitarSample()

This test run to check if the clicking the guitarSample button of the GUI is working

Implementation: Uses FX Robot to startup the stage, and then using the robot to to navigate to the drumSample button and click, then it will convert and verify that the guitar tab was converted



```
void guitarSample(FxRobot robot) {
    robot.clickOn("#helpBar");
    robot.clickOn("#sampleBar");
    robot.moveTo("#drumSample");
    robot.clickOn("#guitarSample");
    robot.clickOn("#convert");
    FxAssert.verifyThat("#errorLabel", LabeledMatchers.hasText("Guitar" + "\n" + "Conversion Complete"));
}
```

Input: N/A

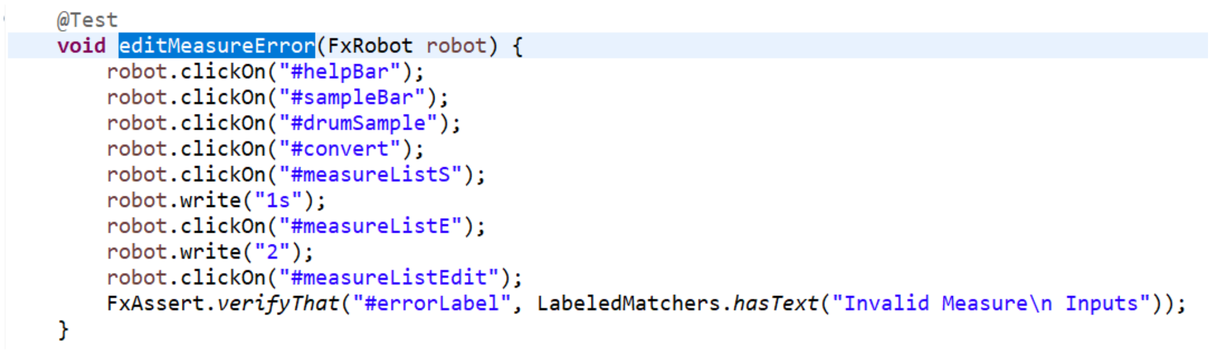
Expected: FX robot verification accepted.

Sufficiency: This test is Sufficient because the program will know there is a guitar sample that users can work off of, this will also verify that guitar conversions the sample tabs also works, as it verifies both using the success label.

Test 5 - editMeasureError()

This test run to check if the clicking the editMeasure error

Implementation: Uses FX Robot to startup the stage, and then using the robot to to navigate to the drumSample button and click, then it will convert and then selects invalid measures and hits editMeasure.



```
@Test
void editMeasureError(FxRobot robot) {
    robot.clickOn("#helpBar");
    robot.clickOn("#sampleBar");
    robot.clickOn("#drumSample");
    robot.clickOn("#convert");
    robot.clickOn("#measureListS");
    robot.write("1s");
    robot.clickOn("#measureListE");
    robot.write("2");
    robot.clickOn("#measureListEdit");
    FxAssert.verifyThat("#errorLabel", LabeledMatchers.hasText("Invalid Measure\n Inputs"));
}
```

Input: N/A

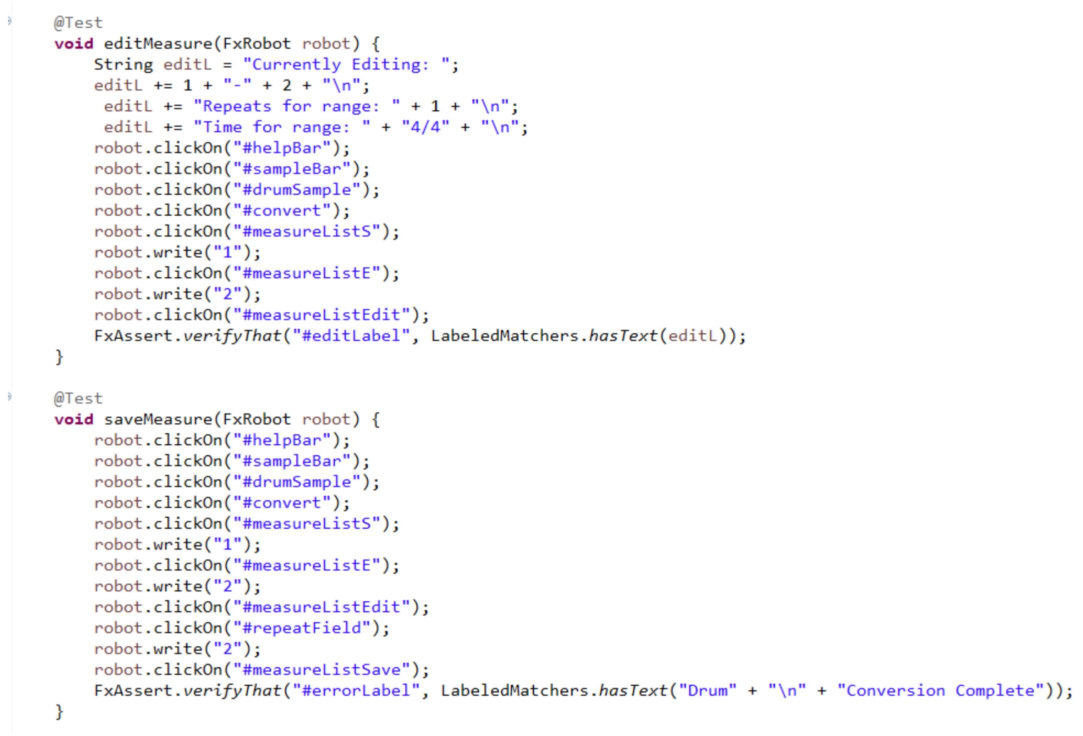
Expected: FX robot verification accepted.

Sufficiency: This test is Sufficient because the program will know if the user messes up a measure input, the program will not fail but instead tell the user.

Test 6 & 7 - editMeasure() & saveMeasure

This test run to check if the clicking the editMeasure process works

Implementation: Uses FX Robot to startup the stage, and then using the robot to to navigate to the drumSample button and click, then it will convert and then selects valid measures and then verify if the information and boxes necessary are provided to user. Then saveMeasure tests if the user enters new info if the tablature is converted properly.



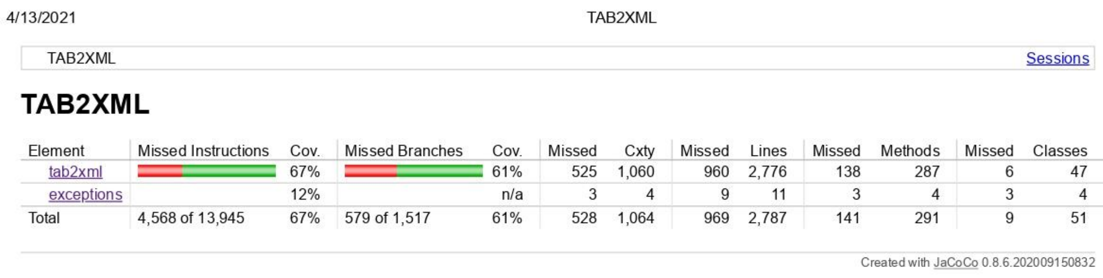
```java
@Test
void editMeasure(FxRobot robot) {
    String editL = "Currently Editing: ";
    editL += 1 + "-" + 2 + "\n";
     editL += "Repeats for range: " + 1 + "\n";
     editL += "Time for range: " + "4/4" + "\n";
    robot.clickOn("#helpBar");
    robot.clickOn("#sampleBar");
    robot.clickOn("#drumSample");
    robot.clickOn("#convert");
    robot.clickOn("#measureListS");
    robot.write("1");
    robot.clickOn("#measureListE");
    robot.write("2");
    robot.clickOn("#measureListEdit");
    FxAssert.verifyThat("#editLabel", LabeledMatchers.hasText(editL));
}

@Test
void saveMeasure(FxRobot robot) {
    robot.clickOn("#helpBar");
    robot.clickOn("#sampleBar");
    robot.clickOn("#drumSample");
    robot.clickOn("#convert");
    robot.clickOn("#measureListS");
    robot.write("1");
    robot.clickOn("#measureListE");
    robot.write("2");
    robot.clickOn("#measureListEdit");
    robot.clickOn("#repeatField");
    robot.write("2");
    robot.clickOn("#measureListSave");
    FxAssert.verifyThat("#errorLabel", LabeledMatchers.hasText("Drum" + "\n" + "Conversion Complete"));
}
```

Input: N/A

Expected: FX robot verification accepted.

Sufficiency: This test is Sufficient because the program will know if the user gets the provided information and if the conversion happens as the user is post conversion editing.

Test Coverage



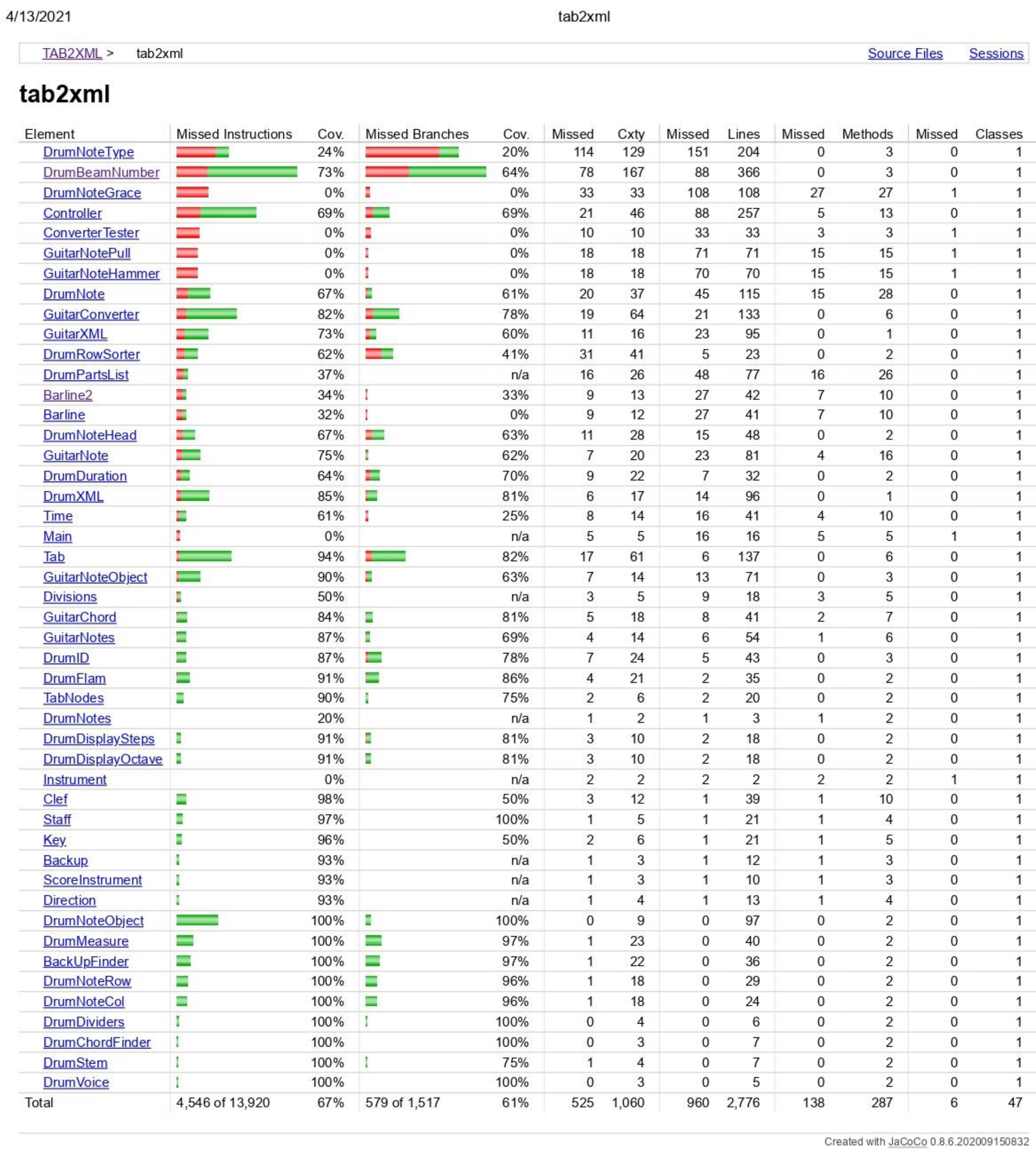
4/13/2021 TAB2XML

TAB2XML Sessions

TAB2XML

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
tab2xml	🟥🟩	67%	🟥🟩	61%	525	1,060	960	2,776	138	287	6	47
exceptions		12%		n/a	3	4	9	11	3	4	3	4
Total	4,568 of 13,945	67%	579 of 1,517	61%	528	1,064	969	2,787	141	291	9	51

Created with JaCoCo 0.8.6.202009150832



4/13/2021 tab2xml

TAB2XML > tab2xml Source Files Sessions

tab2xml

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
DrumNoteType	🟥🟩	24%	🟥🟩	20%	114	129	151	204	0	3	0	1
DrumBeamNumber	🟥🟩	73%	🟥🟩	64%	78	167	88	366	0	3	0	1
DrumNoteGrace	🟥	0%	🟥	0%	33	33	108	108	27	27	1	1
Controller	🟥🟩	69%	🟥🟩	69%	21	46	88	257	5	13	0	1
ConverterTester	🟥	0%	🟥	0%	10	10	33	33	3	3	1	1
GuitarNotePull	🟥	0%	🟥	0%	18	18	71	71	15	15	1	1
GuitarNoteHammer	🟥	0%	🟥	0%	18	18	70	70	15	15	1	1
DrumNote	🟥🟩	67%	🟥🟩	61%	20	37	45	115	15	28	0	1
GuitarConverter	🟥🟩	82%	🟥🟩	78%	19	64	21	133	0	6	0	1
GuitarXML	🟥🟩	73%	🟥🟩	60%	11	16	23	95	0	1	0	1
DrumRowSorter	🟥🟩	62%	🟥🟩	41%	31	41	5	23	0	2	0	1
DrumPartsList	🟥🟩	37%		n/a	16	26	48	77	16	26	0	1
Barline2	🟥🟩	34%	🟥	33%	9	13	27	42	7	10	0	1
Barline	🟥🟩	32%	🟥	0%	9	12	27	41	7	10	0	1
DrumNoteHead	🟥🟩	67%	🟥🟩	63%	11	28	15	48	0	2	0	1
GuitarNote	🟥🟩	75%	🟥🟩	62%	7	20	23	81	4	16	0	1
DrumDuration	🟥🟩	64%	🟥🟩	70%	9	22	7	32	0	2	0	1
DrumXML	🟥🟩	85%	🟥🟩	81%	6	17	14	96	0	1	0	1
Time	🟥🟩	61%	🟥	25%	8	14	16	41	4	10	0	1
Main	🟥	0%		n/a	5	5	16	16	5	5	1	1
Tab	🟥🟩	94%	🟥🟩	82%	17	61	6	137	0	6	0	1
GuitarNoteObject	🟥🟩	90%	🟥🟩	63%	7	14	13	71	0	3	0	1
Divisions	🟥🟩	50%		n/a	3	5	9	18	3	5	0	1
GuitarChord	🟥🟩	84%	🟥🟩	81%	5	18	8	41	2	7	0	1
GuitarNotes	🟥🟩	87%	🟥🟩	69%	4	14	6	54	1	6	0	1
DrumID	🟥🟩	87%	🟥🟩	78%	7	24	5	43	0	3	0	1
DrumFlam	🟥🟩	91%	🟥🟩	86%	4	21	2	35	0	2	0	1
TabNodes	🟥🟩	90%	🟥🟩	75%	2	6	2	20	0	2	0	1
DrumNotes		20%		n/a	1	2	1	3	1	2	0	1
DrumDisplaySteps	🟥🟩	91%	🟥🟩	81%	3	10	2	18	0	2	0	1
DrumDisplayOctave	🟥🟩	91%	🟥🟩	81%	3	10	2	18	0	2	0	1
Instrument		0%		n/a	2	2	2	2	2	2	1	1
Clef	🟥🟩	98%		50%	3	12	1	39	1	10	0	1
Staff	🟥🟩	97%		100%	1	5	1	21	1	4	0	1
Key	🟥🟩	96%		50%	2	6	1	21	1	5	0	1
Backup	🟥🟩	93%		n/a	1	3	1	12	1	3	0	1
ScoreInstrument	🟥🟩	93%		n/a	1	3	1	10	1	3	0	1
Direction	🟥🟩	93%		n/a	1	4	1	13	1	4	0	1
DrumNoteObject	🟩	100%	🟩	100%	0	9	0	97	0	2	0	1
DrumMeasure	🟩	100%	🟩	97%	1	23	0	40	0	2	0	1
BackUpFinder	🟩	100%	🟩	97%	1	22	0	36	0	2	0	1
DrumNoteRow	🟩	100%	🟩	96%	1	18	0	29	0	2	0	1
DrumNoteCol	🟩	100%	🟩	96%	1	18	0	24	0	2	0	1
DrumDividers	🟩	100%	🟩	100%	0	4	0	6	0	2	0	1
DrumChordFinder	🟩	100%		100%	0	3	0	7	0	2	0	1
DrumStem	🟩	100%	🟩	75%	1	4	0	7	0	2	0	1
DrumVoice	🟩	100%		100%	0	3	0	5	0	2	0	1
Total	4,546 of 13,920	67%	579 of 1,517	61%	525	1,060	960	2,776	138	287	6	47

Created with JaCoCo 0.8.6.202009150832