

TAB2XML Testing Document

Group 5

Table of Contents

1.	Introduction	2
2.	JUnit Testing	2
2.1.	Parser	2
2.1.1.	Measure	2
2.1.2.	Pitch	2
2.1.3.	Clef	3
2.1.4.	Parser	4
2.1.5.	Tied	4
2.1.6.	Unpitched	5
2.1.7.	PullOff	5
2.1.8.	Slur	6
2.2.	GUI	7
2.2.1.	DrawSheetLines	7

1. Introduction

This file documents the testing done on the TAB2XML application through the JUnit framework.

2. JUnit Testing

2.1 Parser

This parses the MusicXML string in a format where each element is easily retrievable and able to be used by other classes.

For now, we will be testing the following classes in the Parser package.

For all of the test classes below, two parsers were made; one for wikiGuitarTab and one for wikiDrumTab, in order to test the real, expected input against our cases.

From here on, they will be interchangeably referred to as “input” or “parsers”.

2.1.1 Measure

`testMeasureNumber()` : Tests that the integer returned matches the current measure in the provided input.

Since each input had two measures, in total 4 cases were checked; two measures per parser.

The cases implemented asserted that each measure was being correctly parsed by comparing the expected measure number against the actual measure number returned by `getMeasureNumber()` when called on each of the two parsers.

For example, the expected value of the second measure in the wikiGuitarTab is 2 and so if `wikiGuitarTab.getMeasures.get(1).getMeasureNumber()` returned 2, then the test passed.





`testGetNumNotes()` : Tests that the integer returned matches the expected number of notes from the provided input.

For this test, all the notes in each measure were asserted independently, and thus, there were 4 test cases, two per parser.

The expected number of notes were manually counted from the input and were then compared with the actual integer returned from the `getNumNotes()` method when called on the parsers.

For example, in the wikiGuitarTab, measure one had 8 notes and so this number was compared against the value returned from `wikiGuitarTab.getNumNotes()`. If both the integers matched, then the test passed and the method parsed correctly.

These test cases were sufficient because as can be seen in the screenshot below, the test coverage for the methods in the Measure class was 100%.

		Coverage	Covered Instructions	Missed Instructions: ▾
✓	 Measure.java	 71.2 %	364	147
>	 Measure	 71.2 %	364	147

2.1.2 Pitch

Note: this test is primarily used to test the values for guitar tablature, since drum tablature uses the unpitched class to retrieve the step and octave values.

`testStep()` : Tests if all the expected step values match with the values the parser retrieved from the musicxml, using `getStep()`.

This made sure that the parser was consistently reading the correct note, and to check if the parser was checking the correct step value for that specific note in the tablature.

`testAlter()` : Tests if all of the alter values are correctly read from the musicxml using `getAlter()`, and checks if it matches the corresponding expected values.

This test was crucial because in some scenarios, there would be no alter value present for a specific note, in which we would expect the value returned to be zero if there was none present. This was also important because the alter value signifies if the note was supposed to include an accidental, such as a sharp or flat, which can greatly change how the note sounds.

`testOctave()` : Tests if all of the octave values are correctly read from the musicxml formatted text file, and matches with the values inside the musicxml. This test checks if the octave values matched the expected values from the musicxml, and checks if it is reading the octave value from the correct note in the tablature.

The code coverage for this class can be seen below. As is proven by the metrics, the test cases were sufficient.

	Coverage	Covered Instructions	Missed Instructions: ▾
▾ Pitch.java	100.0 %	30	0
> Pitch	100.0 %	30	0

2.1.3 Clef

`testGetSign()` : Tests if the sign of the clef returned matches the expected sign of the provided instrument.

For this test, two different test cases were tested, depending on the instrument and the measure. For the Guitar, for measure one, the expected value, “TAB”, was compared against the sign returned by `getSign()`.

For the Drum, for measure one, the expected value, “percussion”, was compared against the sign returned by `getSign()`.

For both instruments, the sign of the clef returned by `getSign()` for measure two was compared against “null”, since the second measure shouldn’t have a clef.

`testGetLineValues()` : Tests if the line value of the clef matches the actual placement of clef in the input.

For this test, in terms of measure one, the actual line value of the clef was manually retrieved from the input file and was then compared against the value returned by the `getLineValue` method. For measure two, since there was no clef, the returned value was compared against 0.

These test methods were sufficient because as can be seen below, the code coverage was 100%. Additionally, there are only two kinds of clefs; TAB and percussion, and since we tested for both of them on two different instruments, the testing was deemed sufficient.

	Coverage	Covered Instructions	Missed Instructions: ▾
▾ Clef.java	100.0 %	15	0
> Clef	100.0 %	15	0

2.1.4 Parser

`testGetNumMeasures()` : Tests that the number of measures returned by the `GetNumMeasures` method matches the actual number of measures in the provided input.

To test this method, the expected number of measures were manually retrieved and compared against the integer returned by the method under test.

For example, for the Guitar tab, the number of measures in the file was 2 and so when tested with our method, 2 was also the integer returned. Since the numbers matched, the test was deemed successful and the method correct.

`testGetInstrument()` : Tests that the `getInstrument` method correctly identifies the instrument given in the input.

To test this method, the expected value was once again manually retrieved and then compared against the string returned by the method. Since the assertion returned true and they were indeed the same string, the method was deemed correct.





`testGetArtist()` : Tests that the `getArtist` method correctly retrieves the name of the Artist provided in the input.

For the two parsers that we've used thus far as input, as is obvious, there is no Artist and hence, the method's return value was compared against "null".

`testGetTitle()` : Tests that the `getTitle` method correctly retrieves the Title of the provided input.

Similar to the `getArtist` method, for both the parsers, there was no title element and thus, the method's return value was compared against "null".

The testing was deemed sufficient because of the following code coverage metrics for the Parser class which proved that the majority of the code was covered with our cases:

	Coverage	Covered Instructions	Missed Instructions: ▾
▾  Parser.java	 83.3 %	145	29
>  Parser	 83.3 %	145	29

2.1.5 Tied

`testGetters()` : Tests that the values returned by the `getStart()`, `getStop()`, `getCont()`, and `getLetRing()` methods match the expected values for these elements. For this test, since `wikiGuitarTab` and `wikiDrumTab` had no tied element, instead of using our generic input as we have thus far, a `tiedTest1` parser was used instead.

Although this file had four measures, for testing purposes, in order to save time and still be effective, only the first two measures were considered as input.

To test these methods, the expected value was manually retrieved from the file and then compared with the actual value returned by each method. This was repeated twice, once for each measure.

`testSetters()` : Tests that the values returned by the `getStart()`, `getStop()`, `getCont()`, and `getLetRing()` methods match the expected values once the values are updated by `setStart()`, `setStop()`, `setCont()`, and `setLetRing()`, respectively.

Initially, the values of all the methods were false (this was verified by the test above).

Then, to test the setter methods, the values of all four methods were updated to equal true by their respective setters before the assertion statements. Then, using the getter methods, the values were checked against the new values.

These test methods were deemed sufficient because as can be seen below, the code coverage was 100%

	Coverage	Covered Instructions	Missed Instructions
▼ Tied.java	100.0 %	43	0
> Tied	100.0 %	43	0

2.1.6 Unpitched

`testGetStep()` : Tests that the step returned by the `getStep` method matches the expected step of a note.

To test this method, all step variations were considered through looping if-statements. The step values of all the notes in the `wikiDrumTab` were manually retrieved and compared against the returned value provided by the `getStep` method. This was done for both measures one and two.

`testOctave()` : Tests that the octave returned by the `getOctave` method matches the expected octave of a note.

Similar to `testGetStep()`, to test this method, all octave variations were considered through looping if-statements for both measures in the file. Note that the measures were tested separately in order to keep the testing code easy to read and comprehend. These test cases were deemed sufficient because as can be seen below, the code coverage was 100%.

	Coverage	Covered Instructions	Missed Instructions
▼ Unpitched.java	100.0 %	15	0
> Unpitched	100.0 %	15	0

2.1.7 PullOff

For the following three tests, a `capricho` parser was used, instead of the `wikiGuitarTab` and `wikiDrumTab` parsers since those files had no pull-off elements. Also note that since the `capricho` file was extensive, only the first two measures were considered for testing.

`testGetNumber()` : Tests that the number returned by the `getNumber` method matches the expected pull-off number for the given note.

To test this method, an `ArrayList` was created which stored all the notes from the first measure. Then, while looping through this list, the pull-off number returned by the method was asserted against the expected pull-off number retrieved through manual means. Since the first measure had no pull-off elements for its notes, the returned value was compared with 0.

For measure two, a similar process was followed, but since this measure actually had notes with pull-off elements, each returned value was individually asserted against the expected pull-off number of each note.

`testGetType()` : Tests that the type returned by the `getType` method matches the expected pull-off type for the given note.

To test this method, a similar process to the one above was followed. All measure one notes were asserted against “null” for pull-off types and all notes from the second measure were individually matched against the expected string, which was either “start”, “stop”, or “null” depending on the note under test.

`testGetValue()` : Tests that the value returned by the `getValue` method matches the expected pull-off value for the given note.

As expected, the same procedure was followed for this test as well. All measure one notes were asserted against “null” for pull-off values and all notes from the second

measure were individually matched against the expected string, which was either “null” or “P” depending on whether the note under test had a pull-off value or not. These test cases were deemed sufficient because as can be seen below, the code coverage was 100%.

	Coverage	Covered Instructions	Missed Instructions
▼ PullOff.java	100.0 %	21	0
> PullOff	100.0 %	21	0

2.1.8 Slur

Similar to the PullOffTest class, for the following three tests, a capricho parser was used instead of the wikiGuitarTab and wikiDrumTab parsers since those files had no slur elements. Also note that since the capricho file was extensive, only the first two measures were considered for testing.

`testGetNumber()` : Tests that the number returned by the `getNumber` method matches the expected slur number for the given note.

To test this method, an `ArrayList` was created which stored all the notes from the first measure. Then, while looping through this list, the slur number returned by the method was asserted against the expected slur number retrieved through manual means. Since the first measure had no slur elements for its notes, the returned value was compared with 0.

For measure two, a similar process was followed, but since this measure actually had notes with slur elements, each returned value was individually asserted against the expected slur number of each note.

`testGetType()` : Tests that the type returned by the `getType` method matches the expected slur type for the given note.

To test this method, a similar process to the one above was followed. All measure one notes were asserted against “null” for slur types and all notes from the second measure were individually matched against the expected string, which was either “start”, “stop”, or “null” depending on the note under test.

`testGetPlacement()` : Tests that the value returned by the `getPlacement` method matches the expected slur placement for the given note.

As expected, the same procedure was followed for this test as well. All measure one notes were asserted against “null” for slur placements and all notes from the second measure were individually matched against the expected string, which was either “null” or “above” depending on whether the note under test had a slur or not.

These test cases were deemed sufficient because as can be seen below, the code coverage was 100%.

	Coverage	Covered Instructions	Missed Instructions
▼ Slur.java	100.0 %	21	0
> Slur	100.0 %	21	0

2.2 GUI

This package is quite difficult to test due to its nature and hence the only class that really needed testing is the DrawSheetLines class.

2.2.1 DrawSheetLines

`testGetLine()`: Tests that the coordinates of the line provided are the same coordinates returned by the getter method.

To test this method, we first created a DrawSheetLines object, which was essentially just a line. We then retrieved the individual x and y coordinates through the getter method under test and asserted them with the expected values.


Although 4 assertions were written, this was all just one test case since they all belonged to the same object.

This one test case was sufficient for the entire method because if the getter works for one line, then consequently, it'll work for any and all lines since the only difference between any two lines is their x and y coordinates, which we have already tested.

`testSetLine()`: Tests that the coordinates of the line are successfully updated through the setter method.

To test this method, we first created a DrawSheetLines object (named sheetLine) and initialized it by giving it some random values. Then, we created a new line with new values (named line) through the Line class imported from `JavaFx.scene.shape`. Now, to test the setter method, we updated the values of "sheetLine" to the values of "line" by calling the setLine method on "sheetLine". These values were then asserted with the expected values and if they passed, the testing proved that the setter method worked correctly.

Once again, testing on one line was sufficient because if it works for one line, consequently, it'll work for all other lines as well.

	Coverage	Covered Instructions	Missed Instructions
DrawSheetLines.java	100.0 %	31	0
> DrawSheetLines	 100.0 %	31	0