Allegro Tab Converter

DIVERSIFY MUSIC.

ALLEGRO TAB CONVERTER

TESTING DOCUMENT

Written by:

Mohammed Fulwala

Junhyeong Park

Rafael Dolores

Shawn Verma

Yashraj Rathore

Table Of Contents

Scope and Overview

The Allegro Tab Converter Application is being developed with the goal of converting a music tab into its musicxml format. The software allows the user to input a .txt file that contains a guitar, drum, or bass tab. This can be done by either browsing the .txt file or by simply copying the text into the program. The program will automatically determine the inputted type of instrument and generate the corresponding musicxml file accordingly.

Test Plan

The main approach of testing was through functional testing. The software was tested thoroughly to try to eliminate as many errors as possible that could occur within the current capabilities of the software. Furthermore, the main functionalities of the software that related to running the Software the first time was tested through various test cases. For example, testing if the GUI runs and if the correct buttons are functioning the way they are supposed to. Moreover, the output of each of the instrument parsers were tested to ensure the correct output was generated.

Naming

The naming convention for each test is in the following format:

TestTest1.

Test refers to the name of the method that is being tested and represents an acronym for the   specific functionality being tested. The number represents the number of tests on the specific functionality.

Test Implementation

APP.Java

In this section, we will talk about some tests we implemented.

Test ID	01-01
Test Name	getFileListTest1, getFileList2, getFileList3, getFileList4
Description	Checks if app.java can read input and store it in an array.
Additional Comments	

Test ID	01-02
Test Name	identifyInstrumentTest1, identifyInstrumentTest2, identifyInstrumentTest3, IdentifyInstrumentTest4
Description	Tests if the method can identify guitar, bass, and drum tab.
Additional Comments	

Test ID	01-03
Test Name	guitarTabToXMLTest1, guitarTabToXMLTest2, guitarTabToXMLTest3, guitarTabToXMLTest4
Description	These methods test if the system can correctly generate XML files when a guitar text tablature is provided to it for all four cases.

Additional Comments	Tests for standard tab, multiple collections, time signature customizations, custom tunings

Test ID	01-04
Test Name	bassTabToXMLTest1, bassTabToXMLTest2, bassTabToXMLTest3, bassTabToXMLTest4
Description	These methods test if the system can correctly generate XML files when a bass text tablature is provided to it for all four cases.
Additional Comments	Tests for standard tab, multiple collections, time signature customizations, custom tunings

Test ID	01-05
Test Name	drumTabToXMLTest1, drumTabToXMLTest2, drumTabToXMLTest3, drumTabToXMLTest4
Description	These methods test if the system can correctly generate XML files when a drum text tablature is provided to it for all four cases.
Additional Comments	

Derivation: The derivation of these test cases was based on the core functionality of the program. The method *identifyInstrument* and *getFileTest1* is a core method that is used in app.java and without the proper functionality of this method the program will fail to operate correctly. The method *guitarTabToXML* is a core method that is used in app.java and will ensure that what the program outputs is indeed the correct xml. Therefore, to test these methods test cases were derived to check if this method can correctly identify and output the guitar, drum, and bass instrument tablature xml.

Sufficiency- These test cases are sufficient as they test whether the method can identify the three instruments. The method itself is a basic one and, therefore does not require additional testing. As for the xml output of the instrument, it is also sufficient because each of the xml has a different format to it such as a different type, octave, and step note, to test the accuracy of the method under varying circumstances.

Parsing a Bass Tablature

Test ID	02-01
Test Name	test_durationCount_01, test_durationCount_02, test_durationCount_03, test_durationCount_04.
Description	***test_durationCount*** checks if the method can output the correct duration count that corresponds to the music tab that is being fed into it. In this case we are testing a bass tab.
Additional Comments	N/A

Test ID	02-02
Test Name	test_divisionCount_01, test_divisionCount_02.
Description	**test_divisionCount** checks if the method can output the correct division that corresponds to the music tab that is being fed into it.
Additional Comments	N/A

Test ID	02-03
Test Name	test_stepCount_01, test_stepCount_02.
Description	**test_stepCount** checks if the method can output the correct step count that corresponds to the music tab that is being fed into it.
Additional Comments	N/A

Test ID	02-04
Test Name	test_octaveCount_01, test_octaveCount_02.
Description	**test_octaveCount** checks if the method can output the correct octave count that corresponds to the music tab that is being fed into it
Additional Comments	N/A

Test ID	02-05
Test Name	test_parseAltert_01, test_parseAltert_02.
Description	**test_parseAlter** checks if the method can output the correct alter that corresponds to the music tab that is being fed into it.

Additional Comments	N/A

Test ID	02-06
Test Name	test_tabToCollectiont_01
Description	It checks if the method can store the strings given from the tab into an 2d array list of an array list. it is implemented by having 4 array lists to hold the arrays and a for loop goes through each line and stores them accordingly. Each position of the array list is then compared with the position of the method.
Additional Comments	N/A

Test ID	02-07
Test Name	test_collectionToMeasure_01, test_collectionToMeasure_03, test_collectionToMeasure_03.
Description	Tests a simple bass tab and checks if the output is parsed. It is implemented by having a double for loop that goes through the input. As for the others, the way these test cases are implemented is that the correct duration, division, step count, octave count and alter is stored in a string and is then compared with the actual value the method returns
Additional Comments	Tests for repeated measures with varying inputs.

Derivation- The derivation for these test cases was based on testing the different musical features that are required in order to convert a bass tablature. Furthermore, parsing the required information from a tabalture is key for a successful conversion and therefore *test_tabToCollectiont* and *test_collectionToMeasure* serve these purposes.

Sufficiency- These test cases are sufficient as they test whether the methods can generate the right output. There are multiple test cases covering each aspect of the bass. Each of those test cases have varying inputs in order to cover all the bugs.

Parsing a Guitar Tablature

Test ID	04-01
Test Name	test_durationCount_01, test_durationCount_02, test_durationCount_03, test_durationCount_04
Description	This test case tests the output that is generated from the method *durationCount* in *GuitarParser*.java. It checks if the method can output the correct duration count that corresponds to the music tab that is being fed into it. In this case we are testing a drum tab. The way the test case is implemented is by feeding the guitar tab into an Array list and manually storing the duration count in a variable. The value is then compared with the value from the method.
Additional Comments	

Test ID	04-02

Test Name	test_divisionCount_01, test_divisionCount_01
Description	These test cases test the output that is generated from the method *divisionCount* in *GuitarParser*.java. It checks if the method can output the correct division that corresponds to the music tab that is being fed into it. In this case we are testing a guitar tab. The way the test case is implemented is like the previous one.
Additional Comments	

Test ID	04-03
Test Name	test_stepCount_01, test_stepCount_02
Description	These test cases test the output that is generated from the method step*Count* in *GuitarParser*.java. It checks if the method can output the correct step count that corresponds to the music tab that is being fed into it. In this case we are testing a guitar tab. The way the test case is implemented is that the correct step is stored in a string and is then compared with the actual value the method returns.
Additional Comments	

Test ID	04-04
Test Name	test_octaveCount_01, test_octaveCount_02
Description	These test cases test the output that is generated from the method octaveCo*unt* in *GuitarParser*.java. It checks if the method can output the correct octave count that corresponds to the music tab that is being fed into it. In this case we are testing a guitar tab. The way the test case is implemented is that the correct octave is

	stored in a string and is then compared with the actual value the method returns.
Additional Comments	

Test ID	04-05
Test Name	test_parseAlter_01, test_parseAlter_02
Description	These test cases aim to test the correctness of the parseAlter method in the Guitar. If the expected value is the one that is outputted by the method, then the test cases will pass.
Additional Comments	

Test ID	04-06
Test Name	test_collectionToMeasure_01
Description	This test case aims to check for the correctness of the collectionToMeasure method in GuitarParser. The method itself extracts a specific measure from a collection of measures and places each line of the measure inside an arraylist. In this specific test case, the method is being tested for the sample guitar tablature given in the course website. The expected output is hard coded as entries of a created arraylist. If that arraylist matches the output of the method, it means that the method works and thus, the test case passes.
Additional Comments	

Test ID	04-07
Test Name	test_collectionToMeasure_02
Description	This test case aims to check for the correctness of the collectionToMeasure method in GuitarParser. The method itself extracts a specific measure from a collection of measures and places each line of the measure inside an arraylist. In this specific test case, the method is being tested for guitar tablatures that incorporate forward and backward repeats. The expected output is hard coded as entries of a created arraylist. If that arraylist matches the output of the method, it means that the method works for repeated measures and thus, the test case passes.
Additional Comments	

Test ID	04-08
Test Name	test_collectionToMeasure_03
Description	This test case aims to check for the correctness of the collectionToMeasure method in GuitarParser. The method itself extracts a specific measure from a collection of measures and places each line of the measure inside an arraylist. In this specific test case, the method is being tested for guitar tablatures that incorporate variable number of repeated measures. The expected output is hard coded as entries of a created arraylist. If that arraylist matches the output of the method, it means that the method works for a variable number of repeats thus, the test case passes.
Additional Comments	

Test ID	04-08
Test Name	test_collectionToMeasure_04
Description	This test case aims to check for the correctness of the collectionToMeasure method in GuitarParser. The method itself extracts a specific measure from a collection of measures and places each line of the measure inside an arraylist. In this specific test case, the method is being tested for guitar tablatures that incorporate hammer-ons, pull-offs, and grace notes. The expected output is hard coded as entries of a created arraylist. If that arraylist matches the output of the method, it means that the method works for the mentioned techniques and thus, the test case passes.
Additional Comments	

Derivation:

These test cases were derived based on the functionality of the GuitarParser class. That is, parsing a given tablature by putting each collection of measure into an arraylist, parsing the content of that arraylist to extract a specified measure. Once the measure has been acquired, then the parsing diverts into the smaller details that allocates values to each xml tag. This includes acquiring the step, octave, alter (if there's any), duration, and the type of a note. Based on the aforementioned tasks, specific methods were created to identify these values. To test if the methods acquire the right values, these test cases were created, and they intend to serve as a "control-point" of whether more changes should be implemented in GuitarParser class. As an arraylist was used to contain the parsed data, a for-loop was used to do the actual comparison.

Sufficiency:

These test cases are sufficient as they are able to test each method in GuitarParser with respect to all the cases that might exist. For example, the four collectionToMeasure tests were able to cover four different cases; one case where the tablature contained regular notes with no techniques, a case where there is a variable number of repeats, a case where there is an existing forward and backward repeat, and one case where there is a hammer-on, pull-off, and grace note. Considering every possible case is important to establish sufficiency as they allow you to know if the implemented methods are robust and capable of handling different types of acceptable inputs.

DParser.java (Backend logic for conversion between drum tablature in ASCII format to XML format)

Test ID	05-01
Test Name	test_durationCount_01, test_durationCount_02
Description	Tests the functionality of durationCount method which counts the number of durations of a note and outputs the duration after counting.
Additional Comments	The point on testing this method is to make sure that the bottom line of a measure is never considered during the count and works on dynamic size of input

Test ID	05-02
Test Name	test_durationCountLastLine_01, test_durationCountLastLine_02
Description	Tests the functionality of durationCountLastLine method which is a method specifically for the bottom

	line (bass line) of a measure to do the same functionality as the durationCount method.
Additional Comments	The point on testing this method is to make sure that it only considers the bottom line of a measure while counting the duration.

Test ID	05-03
Test Name	test_divisionCount_01, test_divisionCount_02
Description	Tests the functionality of divisionCount method which is a method that outputs division which is one of the attributes of the drum parser that must be provided in the XML. It takes the size of a measure and time signature as the input
Additional Comments	

Test ID	05-04
Test Name	test_typeDeclare_01, test_typeDeclare_02
Description	Tests the functionality of typeDeclare method which is a method that outputs the type of a note which is a required attribute of a note to parse in a drum parser.
Additional Comments	typeDeclare method serves as an input to typeDeclareGrace method and typeDeclareTwoGrace method

Test ID	05-05
Test Name	test_typeDeclareGrace_01, test_typeDeclareGrace_02
Description	Tests the functionality of typeDeclareGrace method which is a method that outputs the type of a grace note. Grace note is a note that is parsed before a note when a note is a flam.
Additional Comments	Only outputs from the half type of a note down to 16th type of a note

Test ID	05-06
Test Name	test_typeDeclareTwoGrace_01, test_typeDeclareTwoGrace_02
Description	Tests the functionality of typeDeclareTwoGrace method which is a method that outputs the type of two grace notes before a note. Two grace notes are parsed in the XML before the note when the note is a double tap(d).
Additional Comments	Only outputs from the quarter type of a note down to 16th type of a note

Test ID	05-07
Test Name	test_octaveCount_01, test_octaveCount_02
Description	Tests the functionality of octaveCount method which is a method that outputs the octave of the note.
Additional Comments	Software only covers a certain amount of instrument

	IDs and therefore only outputs either a 4 or a 5.

Test ID	05-08
Test Name	test_stepCount_01, test_stepCount_02
Description	Tests the functionality of steCount method which is a method that outputs the step of a note.
Additional Comments	When testing, it must be made sure that all the expectations of output must have a size of 2. ("T" is not allowed, must use "T ")

Test ID	05-09
Test Name	test_beamState_01, test_beamState_02
Description	Tests the functionality of beamState method which is a method that outputs the state of a note in a beam. A beam is a sequence of notes.
Additional Comments	

Test ID	05-10
Test Name	test_beamNumber_01, test_beamNumber_02
Description	Tests the functionality of beamNumber method which is a method that outputs the number of beams that the note that was inputted has. A beam is a sequence of notes.

Additional Comments	

Derivation: The derivation of these test cases was based on the required attributes of the notes in a drum tablature. These methods are called within App.java after taking the user input and deriving the data accordingly to the desired format using the extraction methods (tabToCollection, collectionToMeasure, collectionToMeasureExtractID etc). In other words, these methods do not work on their own without the help of the data extraction methods. The method durationCount is a method used to count the duration of all notes within a measure excluding the bottom line of a measure (the bass). For the bass line, durationCountLastLine method is specifically used to count the duration of the note. divisionCount method is the method to count the division of the measures in the tablature that requires the input of time signature which comes from the user. The allegro Tab Converter supports multiple techniques including grace notes in GuitarParser and similarly, flam and double tab in Drums. The typeDeclare method provides the type of a note which is a required attribute to print in a note and along with that function, typeDeclare is related to the techniques in drums by providing inputs to the typeDeclareGrace method and typeDeclareTwoGrace method that supports the flam and double tap technique. Like the typeDeclare method, octaveCount method, stepCount method. beamState method and beamNumber method also provides required attributes to print on a note.

Implementation- These test cases test the output, a measure of a tablature that is generated from the method *parseDrumMeasure* in App.java. These methods provide the input to the attributes of a measure. The test cases were written to test specific inputs that can be provided to the methods. Parts of War Pigs by Black Sabbath were used as the inputs for the tester methods as the goal of the drum parser for our software is to implement all the functionalities to parse this piece.

Sufficiency- All these tester methods are the minimum number of tests required to test all these methods required to print the attributes. Decent number of methods for the attributes for the note and the measure takes a simpler input with a simpler algorithm and an output and therefore doesn't require that much testing.

Test ID	06-01
Test Name	test_collectionToMeasureExtractID_01, test_collectionToMeasureExtractID_02
Description	Tests one of the core methods that does data handling. collectionToMeasureExtractID extracts the instrument ID from the output of tamboCollection and sorts them in an arraylist of arraylists of string which is the instrument IDs that each measure has.
Additional Comments	CollectionToMeasureExtractID method does not support HF and Hf and deletes them from the arraylist of strings. Therefore, this tester case does not involve the use of HF and Hf.

Test ID	06-02
Test Name	test_collectionToMeasure_01, test_collectionToMeasure_02

Description	Tests one of the core methods that does data handling. collectionToMeasure extracts the measures from the output of tabToCollection and sorts them in an arraylist of arraylists of string which is the arraylists of measures.
Additional Comments	Like the collectionToMeasureExtractID method, collectionToMeasure method also doesn't support HF and Hf and deletes the lines involving them. Testing is done to check that '|' is deleted and measures are separated correctly with no missing elements.

Test ID	06-03
Test Name	test_collectionToMeasureRepeatedMeasure_01, test_collectionToMeasureRepeatedMeasure_02
Description	Tests one of the core methods that does data handling. collectionToMeasureRepeatedMeasure sorts out the measures that are repeated measures, gives them output depending on the situation that the measure is in.
Additional Comments	If a repeated measure is only for a one measure long, this method returns a 4 in an arraylist of Integers in a space corresponding to the measure number. Returns a 1 when the measure is the starting of the repeated measure, a 2 when the measure is the end of a repeated measure and a 3 when the measure is between a starting of a repeated measure and the end of it. Returns a 0 when it's not part of a repeated measure.

Test ID	06-04
Test Name	test_collectionToMeasureRepeatedAmount_01, test_collectionToMeasureRepeatedAmount_02
Description	Tests one of the core methods that does data handling.

	collectionToMeasureRepeatedAmount is a method that allocates the start of the repeated measures the amount of repeats the repeated measure goes through.
Additional Comments	collectionToMeasureRepeatedAmount returns a 0 for the measures that is not the starting of a repeated measure. It returns an integer value and allocates it to an arraylist of integers in a space corresponding to the measure number.

Test ID	06-05
Test Name	test_tabtoCollection_01, test_tabtoCollection_02
Description	Tests one of the core methods that does data handling. tabtoCollection method is the method that the user input goes through to separate them into an arraylist of strings.
Additional Comments	The output of tabtoCollection method is the input for all other data handling methods.

Derivation: These methods are the methods required to extract the desired method from the user input. The derivation of these test cases was based on the functionality of testing the correctness of the user input to data output. The method *tabtoCollection* is a core method which is also the very first method that is executed which sorts the user input into lists of lists of lines of tablature. The rest of the methods sorts those lists of lists to provide desired data to parse. The tester methods for these methods are written to display the functionalities of these methods and to ensure that all the methods work on all specific user inputs.

Implementation- These test cases test the data handling that is generated from the method *drumTabToXML* in App.java. These checks on whether the user input is handled dynamically according to the size of each line of measures and the existence of repeated measures. Drum tab specifically is the tablature that needs to be handled with the consideration of dynamic measure size. The way the test cases are implemented is by creating an array list and manually adding the drum tab lines (lines of measures from War Pig were used in these test cases). This is then followed up by using these methods to compare the output from the method to the desired value.

Sufficiency- These cases are the minimum amount of test cases you need as all the methods output values that are distant from others. They also have many specific inputs they need to handle without errors that must also be tested.

Test ID	07-01
Test Name	test_countBeam_01, test_countBeam_02, test_countBeam_03, test_countBeam_04
Description	Tests the countBeam method which is a method that counts the number of notes in the beam that the measure that is being parsed is dealing with.
Additional Comments	Output goes from 1 to 4

Test ID	07-02
Test Name	test_measureSize_01, test_measureSize_02
Description	Tests the measureSize method that returns the size of a measure + 1 for the other data sorting methods to use.
Additional Comments	Takes the output of tabToCollection method as the input.

Test ID	07-03
Test Name	test_augInput_01, test_augInput_02
Description	Tests the method augInput which is a helper method for the data sorting methods that returns an arraylist of string after passing a parameter of an arraylist of string in an algorithm that reduces the string to take out the instrument IDs
Additional Comments	

Derivation: These methods are the helper methods that are used in other methods for efficiency reasons. Similar to the methods that specify in data handling such as tabtoCollection, these help on data handling, but they do smaller parts of the other methods. These methods don't directly output anything to the user and are invisible to the users. It also outputs data that requires basic understanding to the backend part of the whole project to understand whether it's outputting the right output. That said, the test cases for these methods are somewhat smaller than the other methods and can be tested in a simpler way.

Implementation- These test cases have a simpler pattern of output and handle only a smaller number of inputs. As simple as these methods are, they only have a limited amount of possible test cases and these were all done to make sure that these methods are working properly. As an example, countBeam method only takes an input of a string that represents on what state the note is in and beam number that represents the position of the note within a beam and outputs a new position of the note after being handled by a smaller logic. As this method can only have 4 outputs, all the possible cases were done to test it out.

Sufficiency- The test cases on these methods were done so that all the possible outcomes can be shown. It only has a smaller number of inputs and by that, being efficient.

GUI Testing

Test ID	08-01
Test Name	browseButtonTest()
Description	This method tests if the browse button works.
Additional Comments	I would have tested the file chooser to see if that worked too, but there was no feasible method to check it, which is why this test case only checks if the button works or not.

Test ID	08-02
Test Name	saveButtonTest()
Description	This method checks if the save button is clickable
Additional Comments	This method has the same comments as the browseButtonTest() method

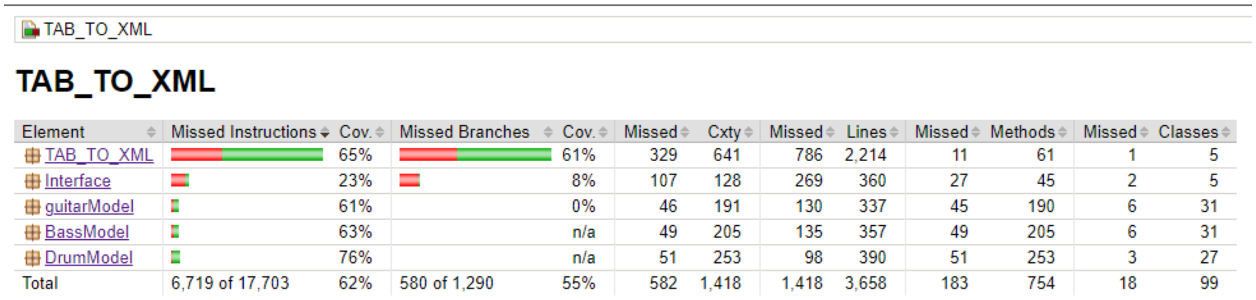
Test ID	08-03
Test Name	exportAndConvertForDrumsTest()
Description	This method checks if the sample drum tab inputted converts and is able to be exported to the device.
Additional Comments	This method was created to check if the parser works for the drums. For this case too, we were unable to check the filechooser because there was no feasible way to do so. Also, sometimes, this test times out due to testFX limitations

Test ID	08-04
Test Name	exportAndConvertForBassGuitar()
Description	This method checks if the sample bass guitar tab inputted converts and is able to be exported to the device.
Additional Comments	This method was created to check if the parser works for the bass guitar. For this case too, we were unable to check the filechooser because there was no feasible way to do so.

Test ID	08-05
Test Name	exportAndConvertForGuitarTest()
Description	This method checks if the sample guitar tab inputted converts and is able to be exported to the device.
Additional Comments	This method was created to check if the parser works for the guitar. For this case too, we were unable to check the filechooser because there was no feasible way to do so. Also, sometimes, this test times out due to testFX limitations

Coverage

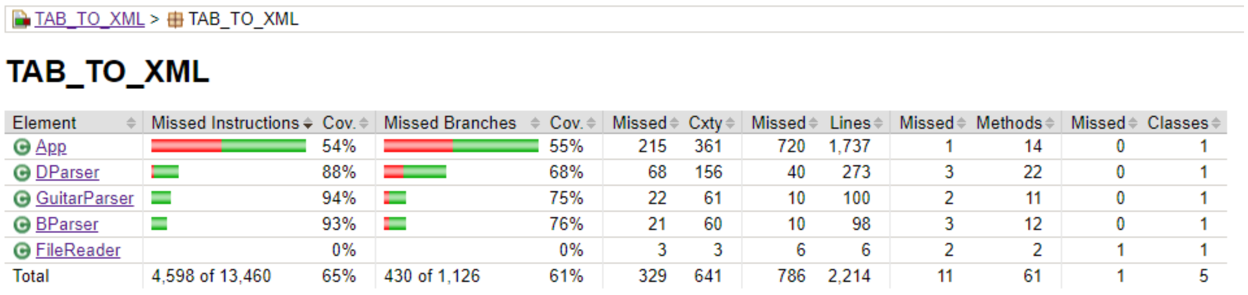
For each varying parameter of the software, there was a test case to test that scenario. The tests check almost every method in the classes that parse the text tablature. Most of the test's pass, some do not due to improper implementation. Almost 70% of the tests pass, and the tests checked were the major methods. Hence, we can say that the tests provided were sufficient.



TAB_TO_XML

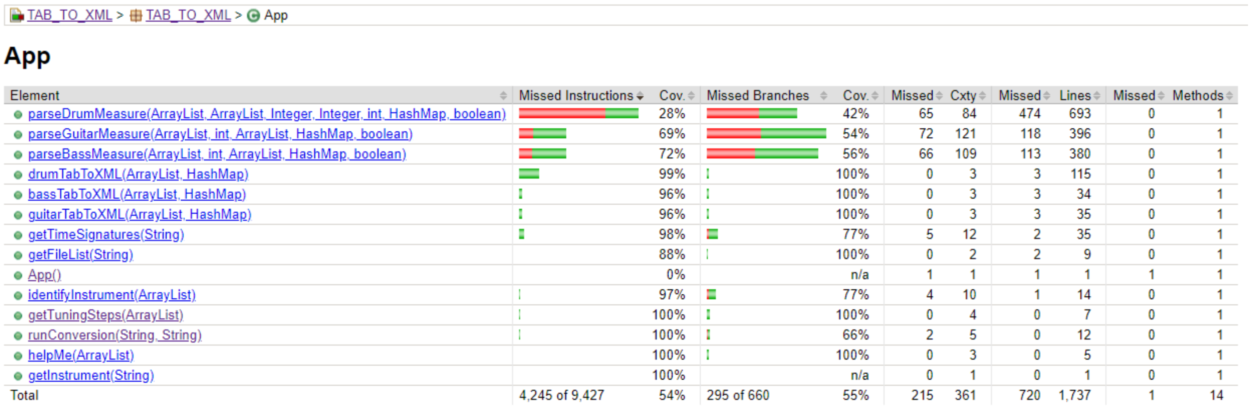
TAB_TO_XML

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
TAB_TO_XML		65%		61%	329	641	786	2,214	11	61	1	5
Interface		23%		8%	107	128	269	360	27	45	2	5
guitarModel		61%		0%	46	191	130	337	45	190	6	31
BassModel		63%		n/a	49	205	135	357	49	205	6	31
DrumModel		76%		n/a	51	253	98	390	51	253	3	27
Total	6,719 of 17,703	62%	580 of 1,290	55%	582	1,418	1,418	3,658	183	754	18	99


TAB_TO_XML > TAB_TO_XML

TAB_TO_XML

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
App	🟥🟩	54%	🟥🟩	55%	215	361	720	1,737	1	14	0	1
DParser	🟩	88%	🟥🟩	68%	68	156	40	273	3	22	0	1
GuitarParser	🟩	94%	🟥🟩	75%	22	61	10	100	2	11	0	1
BParser	🟩	93%	🟥🟩	76%	21	60	10	98	3	12	0	1
FileReader		0%		0%	3	3	6	6	2	2	1	1
Total	4,598 of 13,460	65%	430 of 1,126	61%	329	641	786	2,214	11	61	1	5


TAB_TO_XML > TAB_TO_XML > App

App

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
parseDrumMeasure(ArrayList, ArrayList, Integer, Integer, int, HashMap, boolean)	🟥🟩	28%	🟥🟩	42%	65	84	474	693	0	1
parseGuitarMeasure(ArrayList, int, ArrayList, HashMap, boolean)	🟥🟩	69%	🟥🟩	54%	72	121	118	396	0	1
parseBassMeasure(ArrayList, int, ArrayList, HashMap, boolean)	🟥🟩	72%	🟥🟩	56%	66	109	113	380	0	1
drumTabToXML(ArrayList, HashMap)	🟩	99%	🟩	100%	0	3	3	115	0	1
bassTabToXML(ArrayList, HashMap)	🟩	96%	🟩	100%	0	3	3	34	0	1
guitarTabToXML(ArrayList, HashMap)	🟩	96%	🟩	100%	0	3	3	35	0	1
getTimeSignatures(String)	🟩	98%	🟥🟩	77%	5	12	2	35	0	1
getFileList(String)		88%	🟩	100%	0	2	2	9	0	1
App()		0%		n/a	1	1	1	1	1	1
identifyInstrument(ArrayList)	🟩	97%	🟥🟩	77%	4	10	1	14	0	1
getTuningSteps(ArrayList)	🟩	100%	🟩	100%	0	4	0	7	0	1
runConversion(String, String)	🟩	100%	🟥🟩	66%	2	5	0	12	0	1
helpMe(ArrayList)		100%	🟩	100%	0	3	0	5	0	1
getInstrument(String)		100%		n/a	0	1	0	1	0	1
Total	4,245 of 9,427	54%	295 of 660	55%	215	361	720	1,737	1	14