TAB 2 MUSICXML™

Testing Document

V2 - April 13

By EECS2311 Team 4

- Patchanon Suepai
- Suha Siddiqui
- Nobaiha Zaman Rayta
- Zhilong Lin
- Martin Brejniak

Table Of Contents

1. Introduction

The TAB-2-MusicXML™ has a variety of test cases that have all been implemented using JUnit5 that are used to test our program. This document will cover each one that we currently have, going over what they do, how they were derived, and why they are sufficient.

If you have any further questions or concerns, please feel free to reach out to us through our email (EECS2311Team4@gmail.com).

2. Test Cases

2.1. fileParser()

This function takes a .txt file and attempts to parse it into our internal representation of a guitar note or a drum note.

Test Cases Implemented

1) Given a guitar tab, it asserts an array of guitar notes to be returned.
2) Given a drum tab, it asserts an array of drum notes to be returned.
3) Given an invalid text type, it asserts a FileNotFoundException to be thrown.

These cases were derived from the possible inputs that this function would take, it will always take a txt file that either contains a guitar tab or a drum tab or an invalid file. These cases are sufficient since they test every possible input that this function will reasonably take.

2.2. fileChecker()

This function takes a file path and returns whether or not it is a ".txt" file.

Test Cases Implemented

1) Given a ".txt" file, it asserts a return value of true.
2) Given a non ".txt" file, it asserts a return value of false.

These cases were derived from the possible inputs that this function would take. This function is called when the user selects a file from the file browser so that the user is only able to select a ".txt" file. These cases are sufficient since we only want to check if the user selects a ".txt" or not.

2.3. setMusicNote()

This function takes our internal representation of a guitar note and sets it to its respective musical note.

Test Cases Implemented

1) There is a multitude of test cases that test each possible note from one octave along a guitar. It will take the fret number and string number from the internal representation of a guitar note and asserts the value to be the corresponding musical note.
2) There are also test cases for bass notes, since bass notes are modified guitar notes. If the program flags that this note should be a bass note, the test asserts that the proper bass note is returned rather than a guitar note.

These cases were derived from the possible representation of a guitar note and bass note that we have. Since in a guitar, there are 12 notes that will repeat with a change in octave. These are sufficient since it tests every possible note that the guitar can produce, and the only difference will be the octave changes. Furthermore, bass notes are very similar to guitar notes but shifted. The test cases cover all these scenarios and are sufficient.

2.4. guitarNoteParser()

This function takes a text line that is one line in a guitar/bass tab and returns an array of the notes that are in that guitar/bass tab.

Test Cases Implemented

1) Given a valid tablature line, it asserts an array that contains all the notes inside that line to be returned
2) Given an invalid or empty line, it asserts an empty array to be returned since there are no notes.
3) Given a line that contains a repeated measure, it asserts that the repeat condition is true.

These cases were derived from the possible inputs this function will receive. The text would have already been checked from a previous function so we know that it is definitely a guitar/bass tab. These are sufficient since the only possible inputs that this function can receive is either a tablature line that has notes inside of it or a tablature line that has no notes within it. Additionally, this function also checks for repeated measures, so if given a line with a repeated measure, it will flag that the line contains a repeated measure, so these tests cover all the possible cases.

2.5. drumNoteParser()

This function takes a text line that is one line in a drum tab and returns an array of the notes that are in that drum tab.

Test Cases Implemented

1) Given a valid tablature line, it asserts an array that contains all the notes inside that line to be returned
2) Given an invalid or empty line, it asserts an empty array to be returned since there are no notes.

These cases were derived from the possible inputs this function will receive. The text would have already been checked from a previous function so we know that it is definitely a drum tab. These are sufficient since the only possible inputs that this function can receive is either a tablature line that has notes inside of it, or a tablature line that has no notes within it, so these tests cover all the possible cases.

2.6. guitarXMLParser()

This function takes an array of measures and produces a MusicXML file from it. The function will return null if invalid input is given.

Test Cases Implemented

1) Given a non-empty array of measures, it will successfully produce a music XML file and assert a non-null value to be returned.
2) Given an empty array of measures, it asserts that a null value to be returned, indicating that the tablature could not be parsed.
3) Given a non-empty array of measures with modified notes, such as chord notes, bass note, grace notes, harmonic, etc. assert that these conditions are inputted into the XML.

These cases were derived from the possible inputs this function will receive. The function will always receive an array of measures, with how our program functions, if the parser was able to parse the tablature, the measures array will be non-empty and have valid notes. Otherwise, if the tablature could not be parsed at all, the array will be empty. If given a non-empty measures array, it will definitely produce a music XML file and return non-null, otherwise, it will return null, thus the tests cover all the possible input cases.This parser also checks each note and measure for any special cases such as chords, hammer ons, bass note, grace notes, etc. and the test will check those special cases are added to the XML.

2.7. drumXMLParser()

This function takes an array of measures and produces a MusicXML file from it. The function will return null if invalid input is given.

Test Cases Implemented

1) Given a non-empty array of measures, it will successfully produce a music XML file and assert a non-null value to be returned.
2) Given an empty array of measures, it asserts that a null value to be returned, indicating that the tablature could not be parsed

3) Given a non-empty array of measures with modified notes, such as flam notes, harmonic, etc. assert that these conditions are inputted into the XML.

These cases were derived from the possible inputs this function will receive. The function will always receive an array of measures, with how our program functions, if the parser was able to parse the tablature, the measures array will be non-empty and have valid notes. Otherwise, if the tablature could not be parsed at all, the array will be empty. If given a non-empty measures array, it will definitely produce a music XML file and return non-null, otherwise, it will return null, thus the tests cover all the possible input cases.This parser also checks each note and measure for any special cases such as flam notes, harmonics, etc. and the test will check those special cases are added to the XML.

2.8. start()

This function takes a file path to the txt file that is created when pasting or the user selected ".txt" file from their computer.

Test Cases Implemented

1) Given a non-valid txt file, it asserts that a FileNotFoundException will be thrown.

These cases were derived from the possible inputs the function can receive, it will always receive a file path, if the file path is a ".txt" file, it will proceed to execute the rest of the program. If not, it will throw an error and an error will be presented to the user. Thus these cases are sufficient as they cover all possible outcomes and inputs this function will receive.

2.9. sortNotes()

This function will sort the notes inside our internal representation of a measure, which contains an array of guitar notes. Each guitar note has a position, which is the character number they are in the tablature. Although it is not guaranteed that the guitar note is inserted in the order they are read, thus the need to sort them.

Test Cases Implemented

1) Given two guitar notes that are at position 1 and 2 respectively, once sorted, it will assert that the two notes remain in the same positions.
2) Given two guitar notes that are at position 2 and 1 respectively, once sorted, it will assert that the two notes positions will be switched.
3) Given two guitar notes at position 1 and 1 respectively, once sorted, it will assert that the two notes will remain in the same position.

These cases were derived from the possible representations of our guitar notes. Since this function sorts guitar notes numerically, the only three possible cases are where note 1 is greater than note 2, note 2 is greater than note 1, and when note 1 and note 2 are the same. Thus these test cases are sufficient since it covers all possible scenarios.

2.10. processDuration()

This function processes our internal representation of a guitar note and checks whether or not they are a chord and sets the correct duration for usage in MusicXML.

Test Cases Implemented

1) Given two notes that are in the same position, they will be in a chord, so these tests assert that their durations will be the same and the chord modifier will be set to true.
2) Given two notes that are not in the same positions, these tests assert that their durations will be the difference in positions and the chord modifier will be set to false.

These cases were derived from the need to recognize the correct duration of the note. Depending on whether or not they are in a chord or not makes a big difference, and must be recognized in our to get the correct note duration. These cases are sufficient since they cover all possible scenarios. If two notes are in the same position then they are considered a chord, if they are not in the same position then they are not in a chord.

2.11. setNoteType()

This function processes our internal representation of a note and converts it into musical note types such as “half”, “whole”, “quarter”, etc.

Test Cases Implemented

1) Given a variety of note durations, the tests cover every single note type our program supports, from whole notes all the way to 32th notes. The tests will assert that a note with a number duration will correctly be converted to the text form to be used in musicXML.

These cases were derived from the need to convert the numerical duration of the note into text form to be used in musicXML. These cases are sufficient since they cover all possible scenarios. The tests will check if the numerical note duration properly matches up to the text representation.

2.12. startButtonClick()

This function will check if the robot is pressing in the correct area declared in the GuiWelcome class within the bounds of the ‘Start’ button.

Test Cases Implemented

1) 1Given a set of bounds (x and y integer values) the test will check, through the Robot, if it is moving to the right spot to click on the ‘Start’ button.

The cases were from the need to check the functionality of the GUI itself to check whether or not the button exists in the specified area and can click on the bounds specified.

2.13. confirmButtonClick()

This function will check if the robot is pressing in the correct area declared in the ModificationsPage class within the bounds of the ‘Confirm’ button.

Test Cases Implemented

1) Given a set of bounds (x and y integer values) the test will check, through the Robot, if it is moving to the right spot to click on the ‘Confirm’ button.

The cases were from the need to check the functionality of the GUI itself to check whether or not the button exists in the specified area and can click on the bounds specified.

2.14. convertButtonClick()

This function will check if the robot is pressing in the correct area declared in the GuiUploadWindow class within the bounds of the ‘Convert’ button.

Test Cases Implemented

1) Given a set of bounds (x and y integer values) the test will check, through the Robot, if it is moving to the right spot to click on the ‘Convert’ button.

The cases were from the need to check the functionality of the GUI itself to check whether or not the button exists in the specified area and can click on the bounds specified.

2.15. modButtonClick()

This function will check if the robot is pressing in the correct area declared in the GuiUploadWindow class within the bounds of the ‘Modifications’ button.

Test Cases Implemented

1) Given a set of bounds (x and y integer values) the test will check, through the Robot, if it is moving to the right spot to click on the 'Modifications' button.

The cases were from the need to check the functionality of the GUI itself to check whether or not the button exists in the specified area and can click on the bounds specified.

2.16. editButtonClick()

This function will check if the robot is pressing in the correct area declared in the SaveFile class within the bounds of the 'Edit' button.

Test Cases Implemented

1) Given a set of bounds (x and y integer values) the test will check, through the Robot, if it is moving to the right spot to click on the 'Edit' button.

The cases were from the need to check the functionality of the GUI itself to check whether or not the button exists in the specified area and can click on the bounds specified.

2.17. downloadButtonClick()

This function will check if the robot is pressing in the correct area declared in the SaveFile class within the bounds of the 'Download' button.

Test Cases Implemented

1) Given a set of bounds (x and y integer values) the test will check, through the Robot, if it is moving to the right spot to click on the 'Download' button.

The cases were from the need to check the functionality of the GUI itself to check whether or not the button exists in the specified area and can click on the bounds specified.

2.18. exitButtonClick()

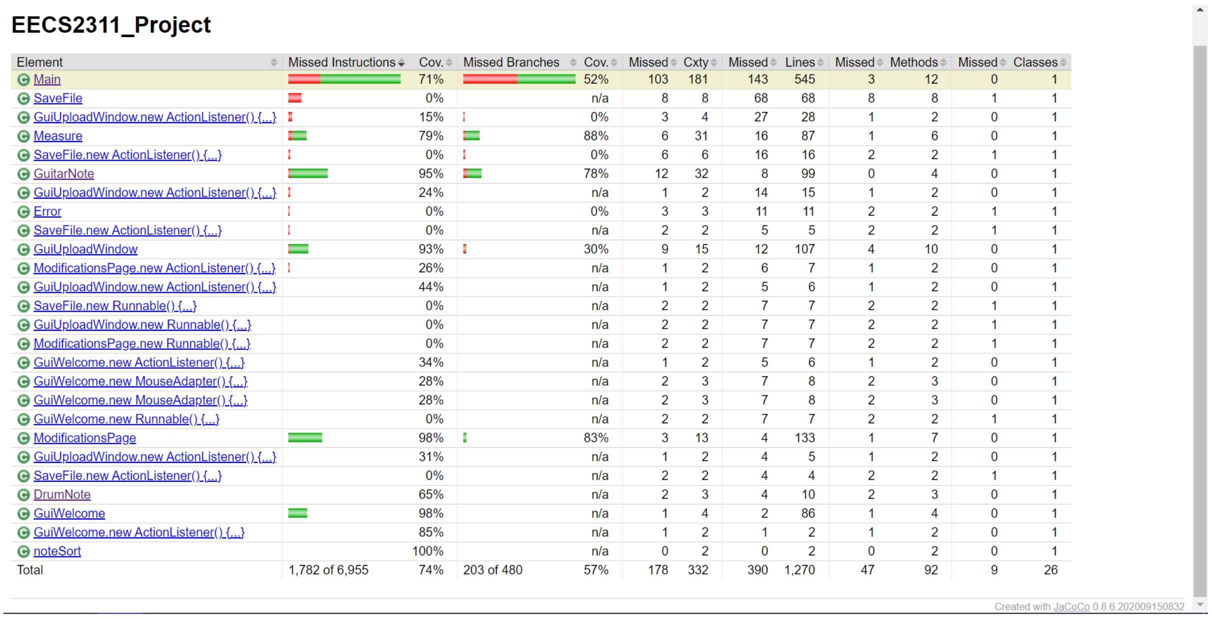
This function will check if the robot is pressing in the correct area declared in the SaveFile class within the bounds of the 'Exit' button.

Test Cases Implemented

1) Given a set of bounds (x and y integer values) the test will check, through the Robot, if it is moving to the right spot to click on the 'Exit' button.

The cases were from the need to check the functionality of the GUI itself to check whether or not the button exists in the specified area and can click on the bounds specified.

3. Test Summary Report



EECS2311_Project

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Main		71%		52%	103	181	143	545	3	12	0	1
SaveFile		0%		n/a	8	8	68	68	8	8	1	1
GuiUploadWindow.new ActionListener() {...}		15%		0%	3	4	27	28	1	2	0	1
Measure		79%		88%	6	31	16	87	1	6	0	1
SaveFile.new ActionListener() {...}		0%		0%	6	6	16	16	2	2	1	1
GuitarNote		95%		78%	12	32	8	99	0	4	0	1
GuiUploadWindow.new ActionListener() {...}		24%		n/a	1	2	14	15	1	2	0	1
Error		0%		0%	3	3	11	11	2	2	1	1
SaveFile.new ActionListener() {...}		0%		n/a	2	2	5	5	2	2	1	1
GuiUploadWindow		93%		30%	9	15	12	107	4	10	0	1
ModificationsPage.new ActionListener() {...}		26%		n/a	1	2	6	7	1	2	0	1
GuiUploadWindow.new ActionListener() {...}		44%		n/a	1	2	5	6	1	2	0	1
SaveFile.new Runnable() {...}		0%		n/a	2	2	7	7	2	2	1	1
GuiUploadWindow.new Runnable() {...}		0%		n/a	2	2	7	7	2	2	1	1
ModificationsPage.new Runnable() {...}		0%		n/a	2	2	7	7	2	2	1	1
GuiWelcome.new ActionListener() {...}		34%		n/a	1	2	5	6	1	2	0	1
GuiWelcome.new MouseAdapter() {...}		28%		n/a	2	3	7	8	2	3	0	1
GuiWelcome.new MouseAdapter() {...}		28%		n/a	2	3	7	8	2	3	0	1
GuiWelcome.new Runnable() {...}		0%		n/a	2	2	7	7	2	2	1	1
ModificationsPage		98%		83%	3	13	4	133	1	7	0	1
GuiUploadWindow.new ActionListener() {...}		31%		n/a	1	2	4	5	1	2	0	1
SaveFile.new ActionListener() {...}		0%		n/a	2	2	4	4	2	2	1	1
DrumNote		65%		n/a	2	3	4	10	2	3	0	1
GuiWelcome		98%		n/a	1	4	2	86	1	4	0	1
GuiWelcome.new ActionListener() {...}		85%		n/a	1	2	1	2	1	2	0	1
noteSort		100%		n/a	0	2	0	2	0	2	0	1
Total	1,782 of 6,955	74%	203 of 480	57%	178	332	390	1,270	47	92	9	26

Created with JaCoCo 0.8.6.202009150832

A summary of the test coverage can be seen above. Overall the test coverage for the program is 74%. This is sufficient as all the important runtime and logic functions are thoroughly tested and most untested parts are GUI components and control flow functions.

Main, in which the majority of the logic resides, has a 71% test coverage, while the remaining untested portion is the control flow function that only calls other functions as the program progresses.

The other classes the program uses such as Measure, GuitarNote, and DrumNote are between 65% and 95% test coverage which we believe is sufficient as it tests all important scenarios that these classes can receive as described in the test cases earlier in this document.