Official Testing Document
MusicXML Converter

EECS2311 Software Development Project

Group 14

Adil Hashmi (adilhashmi.w@gmail.com)
Alexander Arnold (alex290@my.yorku.ca)
Boho Kim (kimboho614@gmail.com)
Kanwarjot Bharaj (kjsingh76@gmail.com)
Stanley Ihesiulo (stanihe1901@gmail.com)

Table of Contents

1. INTRODUCTION

This document will give an overview of all the testing requirements needed to run the MusicXML application successfully. The application is built and designed using Java, Fxml, and Gradle. Compatible with both Eclipse and IntelliJ. The software includes features such as a graphical user interface (GUI), easy to use with an intuitive design. A MusicXML file generator, and finally an end-to-end object-oriented translating algorithm. The algorithm translates tablatures that are user-inputted or entire .txt files. This software is designed for three stages with the first one being to get user input, then translating input and generating MusicXML, and finally creating a new file with the final product ready for use.

2. OBJECTIVES AND TASKS

- **Objectives**

To rigorously test aspects of our code to guarantee error-free results.

- **Tasks**

Create valid test cases to test where the code should be correct.
Create test cases to test the invalidity, where the code should be incorrect.

3. ENVIRONMENT SETUP

3.1 Main Frame

IntelliJ or Eclipse installation is mandatory to run the tests.

4. TESTING STRATEGY

Listed below are the features of our code covered for testing.

4.1 JUnit - Converter_measure_Measure class Test

4.1.1) getNoteQueue() Test

- Testing method: The notes in a measure are played in a certain specified sequence. This test tests if the notes are in the correct sequence, and not out of order. It is a very important test as the order at which notes are played is fundamental to translating a tablature file into an accurate musicXML representation.

- Some notes in a measure are played at the same time as others, forming a chord. This test also confirms that the output of the method getNoteQueue() assigns notes that are the same distance from the start of the measure the accurate distance, as this distance metric will be used in the critical operation of assigning </chord> tags in musicXML to notes which belong in the same chores, as their equal distances will be used to determine if they belong in the same chord.
- Given a measure tab text file:
  Measure tab text file:
  ```
  e|---------------------------------------------------------|
  B|----------6----------------------------------------------|
  G|-2-------------------------------------------------------|
  D|-2----3---5----------------------------------------------|
  A|---0-----------------------------------------------------|
  E|---------------------------------------------------------|
  ```

  Where Note[“2”,1] means a note object with fret 2, and distance 1 from start of measure,

- Example of a valid test:

  String[] {Note[“2”,1], Note[“2”,1], Note[“0”,4], Note[“3”,7], Note[“6”,11], Note[“5”,11]}

- Example of an invalid test:

  String[] {Note[“2”,2], Note[“2”,1], Note[“0”,4], Note[“3”,7], Note[“6”,11], Note[“5”,11]}

4.2 JUnit - Converter_measure_line_GuitarMeasureLine Test

Tests a method in the measureLine class focused on guitar.

4.2.1) createNoteList() Test

- Testing method: The measureLine class has a multitude of methods parsing different aspects within a tab. The following seven JUnit tests focus on the createNoteList method which reads tabs line by line while counting each dash '-', it also extracts all the different types of notes.

- To check if this is implemented properly and gives the correct notes, three 'testValidity()' tests check to see if single notes are extracted from lines of different sizes and the specific instrument notes. Three 'testInvalidity()' tests check to see if the code can catch the incorrect inputs as well, so the code does not just output anything, it follows a specific order. One 'testInvalidNaming' test checks to see if the correct guitar notes are used. An array of non-guitar notes are run through the tester to catch invalidity.

- Example of valid test:
  String s1 = ("---12---6-4-5-8-1-----------2-");
- Example of invalid test:
  String s5 = ("----2----|---4----8--|");
- Example of invalid name test:
  String[] names = {"x", "z", "v", "q", "w", "u", "y", "i", "o", "p", "X", "Z", "V", "Q", "W", "U", "Y", "I", "O", "P"};

  These tests are efficient because they target the core parsing methods which are used to read the .txt files and which are then converted into properly notated MusicXML files.

4.2.2) RegexPattern() Test

- Testing method: Further rigorous testing will be implemented focusing specifically on if the Regex can read more complex patterns correctly, ones that include hammer-ons, pull-offs, slides, bends, releases, etc.

- To check if the Regex can read and give good instructions for constructing XML patterns correctly, other methods in the measureLine() class, and the patterns() class will be tested once complex musical features such as hammer-ons and more are implemented.

- These tests are mandatory in order to play musical notes as intended, songs do not have single, same length notes all around, they differ and have intentional pauses and slides between notes for rhythm. With proper testing, this pattern test will play a crucial part in generating the final MusicXML files.

4.3 JUnit - Converter_MeasureGroup Test

Tests the MeasureGroup class which aggregates Measures to form a score.

4.3.1a) testValidate_validInput1(), testValidate_validInput2(), testValidate_validInput3()

- Testing Method: We pass in various known tabs in the form of an origin (a list of each line of the tab) and confirm that the measure group generated is indeed valid. The lines of the tab must start with valid characters and the tab itself should not have any errors that make it unparseable.
- Example:

  String mg_0 = "[0] e|-3-2-2-0----|";
  String mg_1 = "[1] a|-3-2-2-0----|";
  ...
- Evaluates true if all lines of the measure start with a valid character, are correctly formed, and are preceded by the start index of where they are found in the original root string.

4.3.1b) testValidate_invalidInput1(), testValidate_invalidInput2(), testValidate_invalidInput3()

- Testing method: We pass in known invalid tabs and then confirm that they do not pass the validation test
- Example:

  String mg_0 = "[0] E|------------1-|";
  String mg_1 = "[10] A|-----1-|";
  String mg_2 = "[20] abcd|--2-|";
  ...
- Evaluates false if one or more lines of the measure break the validity rules outlined above.

4.3.2) testToXML()

- Testing method: Create a known valid output XML using a known input, revise the output so that it is accurate, and then compare with future outputs to ensure the underlying mechanism still generates the same XML.
- Reads measureGroupToXMLTest.xml from project resources to compare with the hard coded output.

4.3.3) measureCountValidation()

- Testing method: Ensure that the predefined measure group consists of 6 separate lines after it is parsed (known that it does).

4.3.4) positions()

- Testing method: Ensure that the index of each line (where it is found in the root string) is parsed correctly from the MeasureGroup input

4.4 JUnit - GuitarConverter_GuitarNote Test

It tests some features of note that will be needed to make an XML script.

4.4.1) makeOctave() test

- Testing methods: With given information about fret number and string number of notation, we need to generate an octave corresponding to the specific position. Therefore, we made an octave method to obtain the right octave to convert.
- To check octave, start checking from fret 0 in string 6. Since guitar increases octave from string 6 to 1, set the right octave that we expect and compare it with octave which is generated by octave method using a loop. If the result generated by this method is not the same as the octave which was set, the test is failed.

  Default set octave:

  Fret 0 ~ 7 with String 6 / Fret 0 ~ 2 with String 5 : Octave 2

  Fret 8 ~ 19 with String 6 / Fret 3 ~ 14 with String / Fret 0 ~ 9 with String 4 / Fret 0 ~ 4 with String 3 /Fret 0 with String 2 : Octave 3

  Fret 15 ~ 19 with String 5 / Fret 10 ~ 19 with String 4 / Fret 5 ~ 16 with String 3 / Fret 1 ~ 12 with String 2 / Fret 0 ~ 7 with String 1 : Octave 4

  Fret 17 ~ 19 with String 3 / Fret 13 ~ 19 with String 2 / Fret 8 ~ 19 with String 1 : Octave 5

  This test is sufficient to extract octave information which will be used for XML scripting in the system because the octave is not affected by other musical features. There is no other case of the octave of a guitar which this method can not generate.

4.4.2) makeKey() test

- Testing method: With given information about fret number and string number of notation, we need to generate a key corresponding to the specific position. Therefore, we made a key method to obtain the right key to convert.

  To check the key, make a key array contains all of the keys used in a guitar.

  String[] keys = {"C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "B"};

  Start from fret 0 in String 6. Since Fret 0 in String 6 has key ‘E’, set the first start index 4 to get ‘E’ and start comparing using a loop. Since musical keys are used recursively whenever octave is changed, if the index is out of the boundary of the array, reposition the index to 0 and keep comparing until the last Fret in String 1 (Fret 19 in String 1). Do this step with each String 6 to 1, respectively. If the result generated by the key method is not the same as the key that the index is pointing at, the test is failed.

This test is sufficient to extract key information which will be used for XML scripting in the system because the octave is not affected by other musical features. It is obvious with given fret and string information. There is no other case of the key of a guitar which this method can not generate.

4.4.3) testPitch1() / testPitch2()

- Testing method: With given information about octave and key, we need a proper string that will be used in XML scripting. Therefore, pick some octaves and keys randomly to test. Since there are 2 formats of pitch scripting, we separate it to test, respectively.

  The first group does not contain '#' in the key string. It means the keys are natural notes (e.g white keys in Piano). This pitch just contains step and octave information. The step means key. Set some expected pitch string with randomly picked keys and octave. If the result generated by pitchScript method is not the same as the string that we set, the test is failed.

  The first group contains '#' in the key string. It means the keys have one-half step higher than the natural note (e.g black keys in Piano). This pitch contains step, octave and alter information to express '#' in XML. Set some expected pitch string with randomly picked keys and octave. If the result generated by pitchScript method is not the same as the string that we set, the test is failed.

  This test is sufficient to make a proper pitch script that will be included in XML script because this pitch part is not affected by other musical features and it assumes that the given information of octave and key(extracted by other methods) is right. There is no other case of pitch script of a guitar which this method can not generate.

<u>4.5 JUnit - Converter.ScoreTest</u>

- ScoreTest() - Checking if invalid inputs send an exception. Invalid Input include: empty Strings/null
- Score.getStringFragments() breaks down text into segments where each segment is separated by a blank line. If there is whitespace between them, it keeps the amount in the Integer value of LinkedHashMap and the next string starts after counting the whitespaces.

Example:- Two Strings
abcd
  (3 whitespaces)
efgh
The first string is stored as ‘1=abcd’ and the second string is stored as ‘10=efgh’ because after abcd there is a new line (so 4+1). Then, there are 3 white spaces (5+3). Another new line and then it starts on the 10th element (8+2). The functionality where the index is stored might be removed in future if not needed depending on the direction of how we decide to handle the project.
This test checks if the Score class is handling that correctly.

8. CONTROL PROCEDURES

- **Problem Reporting**

If at any time you (the user) are faced with a problem or error running our application and or tests. Please contact any of the emails given on the first page of this document. We are happy to help!