Testing Document

MXLify

EECS 2311 - Group 9

Zach Ross
Christopher Moon
Erika Grandy
Yasser Al Zahed
Faruq Afolabi

02/17/2021

1. Introduction

The purpose of this document is to describe the testing that is being performed on our software. It describes all the automated test cases that are implemented. It also describes any future testing that is still required.

2. Objectives and Tasks

2.1 Objectives:

Our objectives as a group is to ensure the functionality and performance of our software stays fully operational. By having test cases in place, we can ensure that any new changes do not cause issues, and that our program continues to operate as expected.

3 Test Cases

The following lists all test cases currently being conducted:

3.1 Parser Testing

Test Case: testNoteType()
This method takes a beat note (double) and returns the corresponding note type as a string. This method is being tested to ensure that the correct note type is being returned, and the method can handle the beatNote parameter as decimals or fractions.

Test Steps:

1. Call noteType() with a specific beat note, as a decimal (double).
2. Get returned note type (String), and check if equal to expected.

Test Notes:

1.0, 0.5, 0.25, 1/8, 1/16, 1/32,
1/64, 1/128, 1/256, 1/512, 1/1024

Comments: Method may have errors when called with a fraction, as it uses integer division and will result in beatNote = 0.0.

All Cases Passed

Test Case: testAddTitle()

This method takes the title of the track and inserts it into the *misc* hashmap using key *Title.* This method is being tested to ensure that the title is correctly added, and can be received from the hashmap.

Test Steps:

1. Call addTitle() with the test title.
2. Check that the misc hashmap contains the test title under key *Title.*

Test Titles: “Test Title” , ““

All Cases Passed

Test Case: testAddTabType()

This method takes the tablature type and inserts it into the *misc* hashmap using key *TabType.* This method is being tested to ensure that the type is correctly added and can be received from the hashmap for later use.

Test Steps:

1. Call addTabType() with a test type.
2. Check that the misc hashmap contains the test title under key *TabType.*

Test Time Signatures: “Guitar”

All Cases Passed

Test Case: testAddTime()

This method takes the time signature of the track and inserts it into the *misc* hashmap using key *TimeSig.* This method is being tested to ensure that the time signature is correctly added and can be received from the hashmap for later use.

Test Steps:

3. Call addTime() with the test time.
4. Check that the misc hashmap contains the test title under key *TimeSig.*

Test Time Signatures: “4”, “4/4”

All Cases Passed

3.2 File Generator Testing

Test Case: testFileGenerator()

This constructor initializes the variables and objects needed for file generation. This constructor is being checked to ensure that once a file generator is created, the writer, save file location, preferences, and last used saved folder all exist. Otherwise, the program could crash later on due to a null pointer exception.

All Cases Passed

Test Case: testAddInfo()

This method takes the Title input from the GUI and writes it at the top of the MusicXML file, under tag <work-title>. This method is being tested to ensure
This is important since it allows the reader to provide a title to the tablature you are using as input for the software.

Test Steps:

1. Call addInfo() with the test title.
2. Check that the XML written to the output file is as expected.

Test Titles: “Example Title” , “ “, “”

All Cases Passed

Test Case: testOpenPart()

This method uses the part ID number and writes to the MusicXML file, under tag <part id=....>
This method specifies which instrument is being used for the tablature information, which is useful for multiple instruments.

Test Steps:

1. Call addOpenPart() with the part ID.
2. Check that the XML written to the output file is as expected.

Test Part IDs: 1, 3

All Cases Passed

Test Case: testClosePart()

This method writes to the MusicXML file, closing the part tag.
This method closes the part, completing the information given for the particular instrument.

Test Steps:

1. Call closePart().
2. Check that the XML written to the output file is as expected.

All Cases Passed

Test Case: testAttributes()

This method writes the attributes section to the MusicXML file.
These are the key components of the music sheet as it provides the amount of beats per measure, the duration assigned to a note, the type of clef for the music sheet as well as string tunings and the set of octaves that they are based on.

Test Steps:

1. Call attribute method, with division, key Signature, beat, beat type, clef, array of tunes and array of tune octaves.
2. Check that the XML written to the output file is as expected, based on the inputted values.

Test Attributes:

Division - 2
Key Signature - 4
Beat - 4
Beat Type - 4
Clef - “G”
Tunes - "E", "B", "G", "D", "A", "E"
Tune Octaves - 1, 2, 3, 4, 5, 6

All Cases Passed

Test Case: testAddNote()

This method writes the attributes section to the MusicXML file
This is essential as this method provides the notes to be added to the file.

Test Steps:

1. Call addNote() with string amount, fret, note, note type, duration, octave, dot, and alter.
2. Check that the XML written to the output file is as expected, based on the inputted values.

Test Info:

Case 1

Strings - 1
Fret - 1
Note - “E”
NoteType - “half”
Duration - 1
Octave - 3
Dot - 0
Alter - false

Case 2

Strings - 2
Fret - 3
Note - “G”
NoteType - “quarter”
Duration - 1
Octave - 2
Dot - 0
Alter - false

All Cases Passed

Test Case: testAddChord()

This method adds a chord to the MusicXML file.
Method is important as the chord tag needs to be placed for the notes within the chord or else it will not register as a chord to the MusicXML viewer.

Test Steps:

1. Call addChord with notes, note type, duration, octaves, strings, frets, dots, and alters.
2. Check that the XML written to the output file is as expected, based on inputted values.

Test Info:

Notes - ‘E’, ‘B’
Note Type - “half”
Duration - 1
Octaves - 1, 2, 3
Strings - 1, 2
Frets - 1, 2
Dots - 1, 2, 3
Alters - false, false

All Cases Passed

Test Case: testOpenMeasure()

This method opens the measure in the MusicXML file, based on the specific measure number.
Denotes when to open a new measure for notes to be added within the MusicXML file allowing for new measures to be added.

Test Steps:

1. Call openMeasure() with a measure number.
2. Check that the XML written to the output file is as expected, based on inputted values.

Test Info:

Case 1: Measure Number - 1
Case 2: Measure Number - 4
Case 3: Measure Number - 0

Comments: May want to add checking within OpenMeasure() for measure number being negative or 0.

All Cases Passed

Test Case: testCloseMeasure()

This method closes the measure and score-partwise tags in the MusicXML file.
This method is necessary in order to close off a measure, which is important as each measure denotes a different part of the music sheet.

Test Steps:

1. Call closeMeasure().
2. Check that the XML written to the output file is as expected.

All Cases Passed

Test Case: testEnd()

This method closes the score-partwise tag in the MusicXML file.
We need this in order for the file to be complete and allow for the file to be read.

Test Steps:

1. Call closeMeasure().
2. Check that the XML written to the output file is as expected.

All Cases Passed

Test Case: testCurrentIndent()

The File Generator contains a variable tracking the current indent. This method ensures the correctness of the indent when writing multiple sections to the MusicXML file.
We test this because we want the spacing of the tags to be aligned properly, otherwise the file is not recognized by the MusicXML viewer.

Test Steps:

1. Call addInfo() with a test title.
2. Call openMeasure() with measure number - 1.
3. Check that the XML written to the output file is as expected, comparing two strings to ensure that indentation differences cause a fail.

All Cases Passed

3.3 Tuning Testing

Test Case: testDefaultTuning()

This method determines what the default tuning should be depending on the number of guitar strings.
This method is important to be tested since there is the possibility that tablature without tunings is given therefore the program should be able to recognize that and provide tunings.

Test Steps:

1. Define what the default tunings should be.
2. Check that the default tuning for x number of strings matches what it should be.

Test String Amounts: 5, 6, 7, 8, 9

All Cases Passed

Test Case: testGetNote()

This method returns the note based on string and fret, based on a hashmap created from a file containing tuning information.
This method allows for the other methods to get the actual note which is needed for the other methods in the Parser class and File Generator class to function properly.

Test Steps:

1. Call getNote with different strings and frets.
3. Check that the returned note is correct.

Test Strings and Frets:

String E, Fret 3
String E, Fret 1
String E, Fret 21
String D, Fret 11
String G, Fret 4

All Cases Passed

4. Features To Be Tested in Future

4.1 Parser Testing

Testing will be required on any future additions to our parsing system. This will include systems for detecting the instrument, and any other systems that help parse the tabulator.

Methods to be Tested:

- Parser Constructor

4.2 File Generator Testing

Testing will be required on all our future file generation methods to ensure that they output the correct MusicXML code.

4.3 Tuning Testing

Testing will be required to ensure our tuning system applies the correct tuning parameters to the generated MusicXML.

Methods to be Tested:

- getDefaultTuningOctave
- getOctave
- octaveCheck
- Tuning Constructor

4.4 Warning Testing

Testing will be required to ensure that our warning detactions are working correctly. This will include systems that look for formatting issues or systems that check for user input errors.

5. Features Not To Be Tested

5.1 GUI Testing

The GUI is not easily tested with automation. Although, GUI can still be tested manually by a human quality assurance tester.

6. Why Testing is Sufficient

The goal of these tests are to ensure the core features are working with our software automatically. Every time we make a change to the code base we can run these tests to ensure everything is still working. We are testing the Parser class, the File Generator class and the Tuning class as they are responsible for taking the input and generating the tablature. We tested the key methods in these classes as they are crucial for the program to function properly.