EECS2311: SOFTWARE DEVELOPMENT PROJECT

TAB2XML

Testing Document

April 11, 2022

PREPARED FOR

Vassilios Tzerpos,
Students of Lassonde School of Engineering
&
Musicians

PREPARED BY

Hiba Jaleel - 215735020
Kuimou Yi - 216704819
Kamsi Idimogu - 216880288
Maaz Siddiqui - 216402927

Table of Contents

Introduction

The purpose of this document is to provide general information about the test cases that were generated and used to debug and improve the user-experience for our software, the TAB2MXL. These test cases helped ensure that the TAB2XML software runs successfully.

This application is a java-based program, implementing the graphical user-interface which generates a machine-friendly XML file, along with visual and sound information. The information in the XML file allowed us to create a music sheet generator and player that will be easy to understand by users.

Since the project is based on well-tested starter code, this document will focus on two stages of additional functionality: the music sheet generator and the music player.

Objective and Tasks

The objective of the testing procedures are to ensure that the users will have error-free experiences when they are using the software.

To reach this objective, testing procedures should complete the following tasks:

1. Create test cases which simulate user input and compare that to the correct desired output to ensure that the program will handle user input correctly.
2. Create test cases which target specific parts of the program which may cause errors, so that the program can be designed to handle those errors correctly.

Testing Strategy

General Approach

Our testing is based on the Junit framework and manual validation of results which are not machine friendly.

Our testing strategy include two different types of approach:

a) **The Overall Test Approach**

This approach will consider the whole system as a “black box” where we only know the input and output. Test output will be generated automatically, but the output will be checked manually.

b) **Unit Test Approach**

This approach will break down different parts of that system into “units.” Only the possible inputs and outputs of that specific unit will be taken into consideration. The test output will be generated automatically and validated by the program automatically

Due to limitations and the costs of the program validation, the unit tests will not cover the methods that generate human-friendly information, such as the visual and sound components. In order to conduct the program validation, we are using the Junit test, assertion function, to compare the system output with the desired output.

Overall Test Approach

Music Test

XMLParser Test:

Testing Input:

- All .txt files in the test.resource.system.
- The input can be adjusted by adding a new .txt file or editing the existing .txt file.

Sample Input:

https://pastebin.com/v9dS39kE

Testing Methods:

ParserOutputTest.getSampleString() will get those text files as input and feed them into the XMLParser and write the output as a music String in the output folder.

Expected Result:

MusicString that can be parsed and played by the JFugue player.

Sample Expected Result:

https://pastebin.com/A38EN46U

Visualizer Test

VisualizerOutputTest:

Testing Input:

- All .txt files in the test.resource.system.
- The input can be adjusted by adding a new .txt file or editing the existing .txt file.

Sample Input:

https://pastebin.com/v9dS39kE

Testing Methods:

The program will export the inputted text as a PDF file and store it in the output file.

Expected Result:

Music sheet in PDF format.

Sample Expected Result:

alignmentTest.pdf

GUI Test:

previewWindowsTest

Testing Input:

- All .txt files in the test.resource.system.
- The input can be adjusted by adding a new .txt file or editing the existing .txt file.

Sample Input:

https://pastebin.com/v9dS39kE

Testing Methods:

- FXrobot will click the play button.
- The first run will click the play button with the repeat disabled.
- The second run will click the play button with the repeat enabled.

Expected Result:

- The music plays correctly when the play button is clicked.
- There is no error when the music is played.
- The note being played is being highlighted correctly

Unit Test Approach

Player Test

ParserTest

XMLParser is an important part of the program that converts the musicXML into a music String. XMLParser will be tested to ensure that:

- The resulting music string can be recognized by JFugue
- The resulting music string can produce the right sound.

There are different kinds of parts of code which invoke the music player. The important parts listed below have to be tested:

- The instrument of the note
- The duration of the note. (include note type and dot)
- The pitch of the note
- Chord

Hence, we developed following test case to do the test:

InstrumentConverterTest

In the MusicXML, we will use the instrument ID in the MIDI ID. To use it in the JFugue, we have to convert it into a JFugue ID by a method called getInsturment.

This test will test its correctness.

Testing Input:

Instrument ID in the MIDI ID.

Sample Input:

“P1-I45”

Testing Methods:

Call the getInstrument method and compare the returned string with the expected value.

Expected Result:

- JFugue Instrument ID

Sample Expected Result:

“PEDAL_HI_HAT”

ChordTestPercussion

This test will check the correctness of the notes for chords. Chord is when notes play simultaneously with the notes before it.

Testing Input:

List of Notes; some of them will be a chord.

Sample Input:

- Percussion clef
- Note1 with type quarter and instrument of “P1-I36.”
- Note2 with type quarter and instrument of “P1-I36,” chord
- Note3 with type quarter and instrument of “P1-I36,” chord

Testing Methods:

- Call parser.getNoteDetails and compare the result with the expected results.
- Play it with JFugue.

Expected Result:

- MusicString that can be played by the JFugue player with the right sound.

Sample Expected Result:

"V9 [BASS_DRUM]/0.25+[BASS_DRUM]/0.25+[BASS_DRUM]/0.25 "

ChordTestTab

This test will check the correctness of the note for a chord. Chord is when notes play simultaneously with the notes before it.

Testing Input:

List of Notes; some of them will be a chord.

Sample Input:

- TAB clef
- Note1 with type quarter and Pitch of C4
- Note2 with type quarter and Pitch of C4, chord
- Note3 with type quarter and Pitch of C4, chord

Testing Methods:

- Call parser.getNoteDetails and compare the result with the expected results.
- Play it with JFugue.

Expected Result:

- MusicString that can be played by the JFugue player with the right sound.

Sample Expected Result:

"V1 I25 C4/0.25+C4/0.25+C4/0.25 "

DotTest

This test will check the correctness of the dot. Dot is a kind of notation that affects the note’s duration. Each dot will increase the note's duration by 1/(2^# of dots).

Testing Input:

List of Notes; some of them have dots.

Sample Input:

- TAB clef
- Note1 with type quarter and Pitch of C4
- Note2 with type quarter and Pitch of C4, chord
- Note3 with type quarter and Pitch of C4, chord, one dot

Testing Methods:

- Call parser.getNoteDetails and compare the result with the expected results.
- Play it with JFugue.

Expected Result:

- MusicString that can be played by the JFugue player with the right sound.

Sample Expected Result:

"V1 I25 C4/0.25+C4/0.25+C4/0.375 "

TiedTest

This test will check the correctness of the tie. The tie is a kind of notation that will connect notes together.

Testing Input:

List of Notes; some of them are tied.

Sample Input:

- TAB clef
- Note1 with type quarter and Pitch of C4, has a start tie notation.
- Note2 with type quarter and Pitch of C4, chord, has a start and end tie notation
- Note3 with type quarter and Pitch of C4, chord, has an end tie notation.

Testing Methods:

- Call parser.getNoteDetails and compare the result with the expected results.
- Play it with JFugue.

Expected Result:

- MusicString that can be played by the JFugue player with the right sound.

Sample Expected Result:

"V1 I25 C4/0.25-+C4/-0.25-+C4/-0.25 "

DurationTest

This test will check the correctness of the timing for the entire music.

Testing Input:

Tablature

Sample Input:

https://pastebin.com/CWdwKvcK

Testing Methods:

- Call MXLparser.getFullDurationWithRepeat() to get actual duration.
- Use TemporalPLP in the JFugue to parse musicString to get expected duration.
- Compare actual duration with expected duration.

Expected Result:

- Duration from the Temporal PLP should be the same as Duration for the XMLparser.

Sample Expected Result:

https://pastebin.com/2MLt7Tdk

FirstPositionTest

This test will check if the mapping of the measure number to the music String is correct.

We have a map that maps the measure numbers to a position that this measure first appears in the music String.

Testing Input:

Tablature

Testing Methods:

- Search the musicStrings. If this measure is found, it will not appear in a number set. Put it in the number set. Otherwise, this measure appears more than one time.
- Compare the position in the musicString list to the value in the FirstPosition list.

Expected Result:

- When one measure appears for the first time, the ID in the FirstPosition list must equal to the current position in the musicString list. When one measure appears a second or more time, the ID in the FirstPosition list must not equal to the current position in the musicString list.

Visualizer Test

SelectorTest

VElementTest

VElement is a part of a program that stores information about the graphicContent and represents the sheet music.

VElement will be tested to ensure that:

- Their configuration can be adjusted and updated correctly

initConfigTest

This test will check the correctness of the configuration function.

Testing Input:

A key with 4 random double values with one random boolean value.

Testing Methods:

- initConfig with those 5 values.
- Get values with the keys from the getter methods.
- Compare init values and the returned values.

Expected Result:

- init value and returned value must be the same.

updateConfigTest

This test will check the correctness of the configuration function.

Testing Input:

- A key with 4 random double values with one random boolean value.
- Other random values.

Testing Methods:

- initConfig with those 5 values.
- Update value in the VConfigAble element with updateConfig function.
- Get values with the keys from the getter methods.
- Compare the updated values with the returned values from the getter methods.

Expected Result:

- Updated value and the returned value must be the same.

VisualizerTest

Visualizer is an important method that creates and aligns all VElements.

The Visualizer will be tested to ensure that:

- Their graphic content is aligned correctly with the given setting.

PageAlignmentCheck

This test will ensure that the elements in the page will not exceed the height of the page.

Testing Input:

Tablature

Sample Input:

https://pastebin.com/CWdwKvcK

Testing Methods:

- For each page in the visualizer, compare its actual height with the page's height limit and margin.

Expected Result:

- The page's actual height must be lower than the page's height limit minus the margin.

LineAlignmentCheck

This test will ensure that elements in the line will not exceed the page's width.

Testing Input:

Tablature

Sample Input:

https://pastebin.com/CWdwKvcK

Testing Methods:

- For each line in the visualizer, compare its actual width with the page's width limit and margin.

Expected Result:

- The page's actual height must be equal to the page's width limit minus the margin (difference within 10 pts).

MeasureAlignmentTest

This test will ensure that the measure will be placed right after another measure.

Testing Input:

Tablature

Sample Input:

https://pastebin.com/CWdwKvcK

Testing Methods:

- For each line in the visualizer, for each measure, compare its start position with the last measure's start position + the last measure's width.

Expected Result:

- The measure's start position must equal the last measure's start position + the last measure's width.

ImageResourceHandler Test

ImageResourceHandler is a part of the program that will read image asserts from the disk and send them to the visualizer. The ImageResourceHandler will be tested with the following test cases.

ResourceLoadTest:

The purpose of this test case is to ensure that the ImageResourceHandler will load a .json file which contains the path of image asserts. This test is focused on checking if the imageResourceHandler has loaded the .json file successfully and stored the path of image assert into the designated data structure.

Testing Input:

imageList.Json

// check project’s resource folder

Test Method:

We will check the size of the internal map after loading.

Expected Result:

- There is no error that occurred during the loading.
- Size of the internal map is larger than 0.

ResourceURLNonNullTest

The purpose of this test case is to check whether the path under the .json file has been read successfully after ImageResourceHandler loads the .json file.

Testing Input:

imageList.Json

// check project’s resource folder

Test Method:

We will check if the value in the map is not null.

Expected Result:

- There is no error that occurred during the loading.
- The value in the map is not null (String URL).

ResourceNonNullTest

The purpose of this test case is that after ImageResourceHandler loads the path of assert, ImageResourceHandler also responds by loading the image data from the assert path. We will check if the loaded file is not null too.

Testing Input:

imageList.Json

Graphic Assets

// check the project’s resource folder

Test Method:

We will check to ensure that the returned image from the getImage method is not null.

Expected Result:

- There is no error that occurred during the loading.
- The value from getImage is not null (image object).

resourceNullTest

The purpose of this test case is to ensure that the resourceHandler is establishing one-to-one mapping on the assert and its ID.

Testing Input:

imageList.Json

Graphic Assets

// check the project's resource folder

Test Method:

Get an image with ID that did not exist in the .json file

Expected Result:

- There is no error that occurred during the loading.
- The value from getImage is not null (image object).

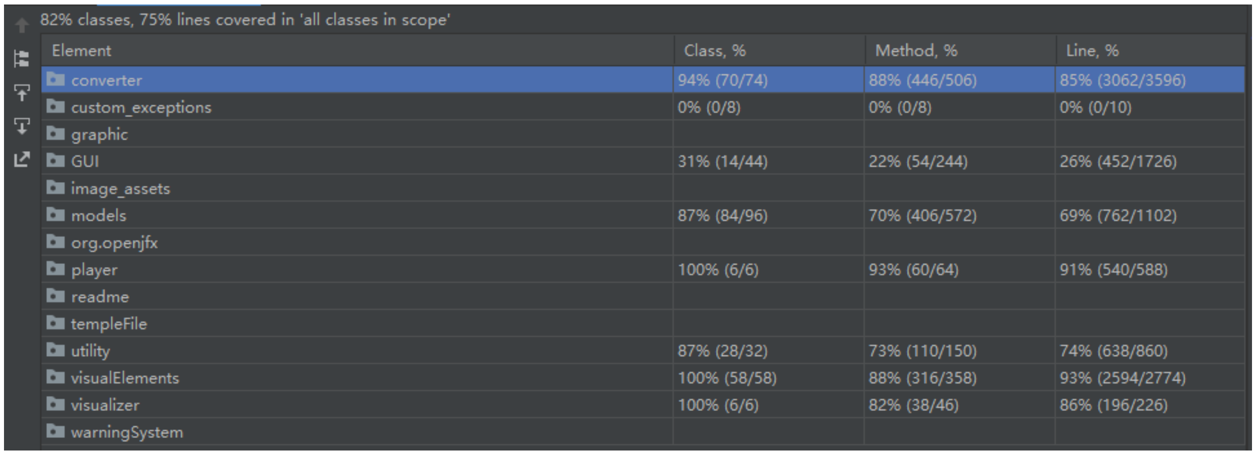
Code Coverage Rate Report

As shown in the image below, our test cases cover over 88% of methods in the VElement package, over 82% of methods in the Visualizer package and over 93% of methods in the Player package.

100% of our classes (added after the starter's code) are covered by our test cases.

82% of classes and 75% of lines are covered overall.

This is sufficient for programmers to identify problems in the program and provide a solid and stable experience to the end-user.



82% classes, 75% lines covered in 'all classes in scope'

Element	Class, %	Method, %	Line, %
converter	94% (70/74)	88% (446/506)	85% (3062/3596)
custom_exceptions	0% (0/8)	0% (0/8)	0% (0/10)
graphic			
GUI	31% (14/44)	22% (54/244)	26% (452/1726)
image_assets			
models	87% (84/96)	70% (406/572)	69% (762/1102)
org.openjfx			
player	100% (6/6)	93% (60/64)	91% (540/588)
readme			
templeFile			
utility	87% (28/32)	73% (110/150)	74% (638/860)
visualElements	100% (58/58)	88% (316/358)	93% (2594/2774)
visualizer	100% (6/6)	82% (38/46)	86% (196/226)
warningSystem			

Full coverage rate report can be found at the link below:

https://github.com/CCSCovenant/TAB2XML/blob/master/Documents/2311CoverageReport.zip