Testing Document

MXLify

EECS 2311 - Group 9

Zach Ross
Christopher Moon
Erika Grandy
Yasser Al Zahed
Faruq Afolabi

04/13/2021

Table of Contents

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. Testing Objective

We want 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.

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 can also produce a test coverage report to ensure that most of the code base is being tested.

3 Back-End Test Cases

The following lists the important test cases currently being conducted within the MXLify system:

3.1 String Parser Testing

Test Case: testParser()
This method runs a full string parser test using a sample tablature input and checks the generated MusicXML file for correctness.
It is important for this system to be tested to ensure that the string parser is fully operational and is working correctly.

Test Steps:

1. Define the tablature input.
2. Define the tablature instrument.
3. Pass the input and instrument into the StringParser.

Test info:
The generated MusicXML file should match the expected MusicXML result

All Cases Passed

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:

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

Test Time Signatures: “4”, “4/4”

All Cases Passed

3.2 Drum Parser Testing

Test Case: test()

This method runs a full drum parser test using a sample tablature input and checks the generated MusicXML file for correctness.
It is important for this system to be tested to ensure that the drum parser is fully operational and is working correctly.

Test Steps:

1. Define the tablature input.
2. Define the tablature instrument.
3. Pass the input and instrument into the DrumParser.

Test info:
The generated MusicXML file should match the expected MusicXML result

3.3 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, alter, hammerons, harmonic, and grace.
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
hammerStart -false
hammerContinue-false
hammerDone-false
Harmonic-false
grace-false

Case 2

Strings - 2
Fret - 3
Note - “G”
NoteType - “quarter”
Duration - 1
Octave - 2
Dot - 0
Alter - false
hammerStart -false
hammerContinue-false
hammerDone-false
Harmonic-false
grace-false

Case 3

Strings - 1
Fret - 1
Note - “E”
NoteType - “half”
Duration - 1
Octave - 3
Dot - 0
Alter - false
hammerStart -false
hammerContinue-false
hammerDone-false
Harmonic-false
grace-true

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

3.4 String 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.
2. 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

3.5 Drum Tuning Testing

Test Case: testDrumSupport()
This method is used to test whether we support a particular drum or not, based on it name e.g. BD for Bass Drum, HH for Hi-hat or S for Snare.

Test Steps:
1. Call the drumSupportCheck method
2. Check that it returns ‘true’ or ‘false’ in the right cases

Test Drums: C, HH, BD, X, Hf, C2

All Cases Passed

Test Case: testGetNote()
This method returns the note mapped to a specific drum based on the drum name provided. It is important to test to make sure that the right notes are used in the MusicXML of the drum tablature.

Test Steps:
1. Call the getNote method
2. Check that it returns the right note mapped to the drum specified, or ‘null’ if the drum is not supported

Test Drums: BD, FT, SN, T2, HT, Rd, HH, C, C2, Hf, X

All Cases Passed

Test Case: testGetOctave()
This method returns the octave mapped to a specific drum based on the drum name provided.
It is important to test to make sure that the right octave numbers are used in the MusicXML of the drum tablature.

Test Steps:
1. Call the getOctave method
2. Check that it returns the right octave number mapped to the drum specified, or -1 if the drum is not supported.

Test Drums: BD, FT, SN, T2, HT, Rd, HH, C, C2, Hf, X

All Cases Passed

Test Case: testGetID()

This method returns the ID mapped to a specific drum and symbol based on the drum and symbol provided e.g. ‘HH’ and ‘o’ for open Hi-hat, or ‘HH’ and ‘x’.
It is important to test to make sure that the right ID is used in the MusicXML of the drum tablature.

Test Steps:

1. Call the getID method
2. Check that it returns the right ID based on the drum and symbol provided or ‘null’ if the drum is not supported

Test Drums and Symbols:
‘BD’ and ‘o’, ‘FT’ and ‘o’, ‘SN’ and ‘o’, ‘T2’ and ‘o’, ‘HT’ and ‘o’, ‘Rd’ and ‘x’, ‘HH’ and ‘o’, ‘HH’ and ‘x’, ‘C’ and ‘x’, ‘C2’ and ‘x’, ‘Hf’ and ‘x’, ‘K’ and ‘o’.

All Cases Passed

Test Case: testGetVoice()

This method returns the voice of a specific drum based on the drum provided.
It is important to test to make sure that the drums are put in the right voices in the MusicXML output.

Test Steps:

1. Call the getVoice method
2. Check that it returns 2 if it’s a Bass drum, or 1 if it’s any other drum and -1 if the drum is not supported

Test Drums: BD, FT, SN, T2, HT, Rd, HH, C, C2, Hf, X

All Cases Passed

3.6 Save & Load Testing

Test Case: testSave()

This method saves text tablature and its metadata to a .mxlify save file and then reads its contents to ensure correctness.
It is important for this system to be tested to ensure that .mxlify files are being saved correctly.

Test Steps:

1. Define what is in the text area as input.
2. Define all the metadata
3. Pass the input and metadata into the SaveManager.

Test info:
The generated file should match the expected result

All Cases Passed

Test Case: testLoad()

This method loads a .mxlify save file and ensures that the data is loaded into the system correctly.
It is important for this system to be tested to ensure that .mxlify files are being loaded correctly and all the data is being loaded into the system.

Test Steps:

1. Define the .mxlify save file location.
2. Pass the save file location into the LoadManager.

Test info:
The text area should be populated with the loaded tablature. The metadata should populate all the input fields (Instrument, song title, time signature, measure edit)

All Cases Passed

3.7 Format Checker Testing

Test Case: testFormat()

This method generates a formatting error depending on the input formatting.
It is important for this system to be tested to ensure that this formatting error continues to be detected correctly.

Test Steps:

1. Define what is in the text area as input.
2. Pass the input and instrument type into the FormatChecker.

Test info:
Case 1: Empty input results in format error type 2
Case 2: Random text input results in format error type 2
Case 3: Correctly formatted input results in format error type 0
Case 4: Slightly incorrectly formatted input results in format error type 1
Case 5: Incorrect tune input results in format error type 2
Case 6: Incorrect octave input results in format error type 2

All Cases Passed

4. Front-End Test Cases

4.1 GUI Panel Testing

4.1.1 FileDropPanelTest

Test Case: testElements()
This test checks that all fields exist once a FileDropPanel is created.

Test Steps:
1. Create FileDropPanel.
2. Test that dropFilePath, DropPanel, DropLoc, and DropLabel are all not null.

All Cases Passed

Test Case: testDropLabel()
This test checks that all properties of the DropLabel are correct.

Test Steps:
1. Create FileDropPanel.
2. Get DropLabel from FileDropPanel.
3. Check DropLabel for correct instance, colours, alignment, and text.

All Cases Passed

Test Case: testDropLoc()
This test checks that all properties of the DropLoc are correct.

Test Steps:
1. Create FileDropPanel.
2. Get Dropocrom FileDropPanel.
3. Check DropLoc for the correct instance, height, background colour, and that the image exists.

All Cases Passed

4.1.2 TextInputTest

Test Case: testGetInput()
This method takes a string array of input and converts it into a double array list of strings for later parsing.

Test Steps:
1. Set text in text field.
2. Call getInput() with a sample string array.
3. Check that the returned double array list of strings is as expected.

Test Info:
Adding text to the textArea will allow the conversion to be done. An empty text area is expected to return null.

All Cases Passed

4.1.3 MyFrameTest

Test Case: testPanels()
This test checks that all panels belonging to MyFrame exist once MyFrame is created.

Test Steps:
1. Check mainContentPanel is not null.
2. Check textInputContentPanel is not null.
3. Check fileUploadContentPanel is not null.

Test Info:
Any of these panels being null would mean GUI was not created properly, and would render the program unusable.

All Cases Passed

Test Case: testTitle()
This test checks that the MyFrame has the correct title

Test Steps:
1. Check the title on MyFrame is “MXLify”

Test Info:
While this test is not crucial, it is important that a displayed title has the correct spelling of the software name.

All Cases Passed

Test Case: testSize()

This test checks that the MyFrame is the correct size

Test Steps:

1. Get width of MyFrame, check that it is 1280
2. Get height of MyFrame, check that it is 720

Test Info:

This test ensures that the program is always the correct ratio, as the MyFrame adjusting sizes cause issues with panel, button, and label placement.

All Cases Passed

4.2 GUI PopUp Testing

Test Case: testClearPopUp()

This test checks that the clear popup is created correctly, along with its buttons and visual aspects.

Test Steps:

1. Create new clearPopUp.
2. Check that the popup and it's buttons exist.
3. Check that the state of the program is *In Popup.*
4. Check button background colour.
5. Click the no Button (Using automation).
6. Check that the state of the program is *Not In Popup.*

Test Info:

This test ensures that the popup is created, and that the state of the program functions as expected. The popup closing but the program state not changing would cause issues in other areas of the program.

All Cases Passed

4.3 Automated GUI Testing

Test Case: testClearAccept()

This test uses a Java Swing Robot to perform automated GUI testing. It checks that text within the text area is erased when the clear button is pressed. By extension it also checks that the clearPopUp functions properly, when a user accepts to erase the text area.

Test Steps:

1. Add text to the text area.
2. Check clear button colour (Mouse not over button).
3. Click the clear button.
4. Check clear button colour (Mouse hovering over button).
5. Click **yes** button on ClearPopUp.
6. Check that text has been cleared from the text area.

.

Test Info:
This test is specifically for the case when the user **does** want to clear the text area, and confirms it within the popup. Through testing the colour change in the button, the actionEvents for the mouse location are being tested in addition to button clicks.

All Cases Passed

Test Case: testClearDeny()

This test uses a Java Swing Robot to perform automated GUI testing. It checks that text within the text area is not erased when the user denies to clear in the clear popup. By extension it also checks that the clearPopUp functions properly, when a user denies to erase the text area.

Test Steps:

1. Add text to the text area.
2. Check clear button colour (Mouse not over button).
3. Click the clear button.
4. Check clear button colour (Mouse hovering over button).
5. Click **no** button on ClearPopUp.
6. Check that text has not been cleared or modified in the text area.

.

Test Info:
This test is specifically for the case when the user **does not** want to clear the text area, and denies the clear within the popup. Through testing the colour change in the button, the actionEvents for the mouse location are being tested in addition to button clicks.

All Cases Passed

Test Case: testClearEmpty()

This test uses a Java Swing Robot to perform automated GUI testing. It checks that text within the text area is erased when the clear button is pressed. By extension it also checks that the clearPopUp functions properly.

Test Steps:

1. Ensure the text area is empty.
2. Check clear button colour (Mouse not over button).
3. Click the clear button.
4. Check clear button colour (Mouse hovering over button).
5. Check that the clearPopUp was not created.

.

Test Info:
This test is specifically for the case when the text area contains no text. As there is no text to be cleared, clicking the clear button should not result in a popup.

All Cases Passed

The GUI is not easily tested with automation. We are doing some simple GUI testing, but the remainder of the GUI can be tested manually by a human quality assurance tester. GUI automation testing is being done on the clear button and text area. Other GUI features are implemented in a similar way.

5. Testing Sufficiency

5.1 Testing Coverage Report

Overall coverage:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
tab2mxl		80%		67%	310	674	434	2,372	9	95	1	11
gui		69%		35%	124	183	274	622	49	92	0	12
gui_panels		68%		15%	62	96	159	492	19	50	0	13
gui_popups		53%		20%	17	29	100	213	13	24	3	9
Total	4,845 of 19,429	75%	572 of 1,421	59%	513	982	967	3,699	90	261	4	45

Front end coverage:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
TextInputContentPanel		68%		21%	35	45	91	263	5	12	0	1
FileUploadContentPanel		65%		0%	10	12	32	93	1	3	0	1
TextInputContentPanel.new MouseAdapter() {...}		12%		n/a	2	3	6	7	2	3	0	1
TextInputContentPanel.new MouseAdapter() {...}		12%		n/a	2	3	6	7	2	3	0	1
TextInputContentPanel.new MouseAdapter() {...}		12%		n/a	2	3	6	7	2	3	0	1
FileUploadContentPanel.new MouseAdapter() {...}		12%		n/a	2	3	6	7	2	3	0	1
TextInputContentPanel.new ActionListener() {...}		25%		0%	5	6	5	6	1	2	0	1
FileDropPanel.new MouseAdapter() {...}		26%		n/a	2	3	4	5	2	3	0	1
MyFrame.ImagePanel		82%		n/a	1	3	2	13	1	3	0	1
MyFrame		99%		n/a	0	2	0	34	0	2	0	1
TextInputContentPanel.new DocumentListener() {...}		97%		n/a	1	4	1	8	1	4	0	1
FileDropPanel		100%		n/a	0	6	0	43	0	6	0	1
TextInputContentPanel.new MouseAdapter() {...}		100%		n/a	0	3	0	7	0	3	0	1
Total	798 of 2,532	68%	78 of 92	15%	62	96	159	492	19	50	0	13

Back end coverage:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
DrumParser		74%		64%	98	193	128	551	0	22	0	1
StringParser		84%		73%	96	230	87	543	1	17	0	1
CreateScore		0%		0%	10	10	48	48	3	3	1	1
FileGenerator		90%		78%	20	80	66	748	1	22	0	1
LoadManager		64%		55%	21	33	17	76	0	4	0	1
StringTuning		90%		57%	27	49	35	135	1	9	0	1
SaveManager		60%		40%	5	6	12	42	0	1	0	1
FormatChecker		90%		77%	14	38	8	127	0	5	0	1
Main		14%		0%	5	7	12	15	2	4	0	1
InstrumentDetection		42%		33%	9	11	13	24	1	2	0	1
DrumTuning		96%		92%	2	19	4	67	0	6	0	1
Total	2,311 of 11,857	80%	367 of 1,143	67%	307	676	430	2,376	9	95	1	11

5.2 Why Testing Is Sufficient

Our overall testing coverage is 75% and our backend testing coverage is 80%. This means that the majority of our code base is being tested with a focus on the backend. The backend is the most important part of our system to be tested since this is where all the logic of the system is and it cannot be easily tested by quality assurance testers.

The front end of the system has a lower coverage of 68% but this is sufficient since it is much harder to test the front end GUI automatically and it can instead be tested via human quality assurance testers. Testing was performed on the main frame and panels to ensure everything exists, and was the proper size and colour. Automated testing was performed on the clearing text area sequence. By extension testing was done on actionEvents for button clicks and mouse location. This allows for the testing to be sufficient as any untested actions were created in similar ways.

These test coverage results show that the most important parts of our system are being tested to ensure that the system remains functional and performant. Since the most important parts are being tested ensuring any bugs or issues are caught, this systems testing is sufficient.