TAB2MXL: Txt to MusicXML File Converter

Testing Document

Monday April 19, 2021

EECS 2032 - Professor Tzerpos

Group 12
Sara Araibi (215700255)
Vishwa Perera (216155947)
Savneet Gill (217386400)
Kaneez Fatima (215711534)

Table of Contents

1.0 Test Case Analysis

1.1 Application and Software System Overview

The results of the test cases reflect on our software system and application of the project. The TAB2MXL takes a music tablature in the format of a text file and converts it to an MusicXML file. Our application is a Java application where it will be running through a gradle task in eclipse. It has features including where the application can save the file, add the title and so on.

1.2 Test Case Scope

The scope of our test cases defines on what method the test case is testing. There are different functions that are used in the scope of the test case. The test cases demonstrated the different methods that is being tested for the product of the customer. These areas that have been tested are a crucial part of the software system as to it shows the customer that the software system is being run to the expectation of the back-end user. Majority of the methods written in the project are ensured that it has been tested to ensure that the method can read the octave, step, alter and so on. These are crucial part of the project for the tablature of the guitar and bass instrument.

2.0 Test Case Design

When designing test cases, there are various inputs and expected outputs in relation to our project. The test case design is using JUnit library in part of the testing design. The inputs in some of our test cases are designs that it is comparing the expected values to the text file being called. Such inputs help ensure that we can compare the output of the text file to the designated output of what the user needs. Test cases include different kind of functions that we are using such as the assertEquals, scanners, assertTrue and so on. These functions are essential part of the test case implementation as to it help us compare, checks if the output is true and so on.

Test cases designs are implements for drums, bass and guitar instruments in a tablature. The test case contains Junit tests to see when uploading a tablature into the GUI and converting it to the desired XML file, test cases were implemented to check weather if the right notes, duration and so on is being read from the code.

3.0 Test Case Description Back End

3.1 TestPitch.java Class

TestPitch.java					
Test Case Names	Expected	Actual	Results (Pass/Fail)	Why/How it was derived?	How it was implemented?
testStep_Normal	"C"	"C"	PASS	The test case was derived to check to see if the expected output which is the character, "C", to the actual output of the variable p. The variable p stores the new note.	It was implemented by using String called expected where it stored the expected character. The Pitch class is getting called by the variable p where it is storing a new note. The expected variable and the p.getStep() is then being compared by using the function assertEquals.
testStep_Sharp	"C"	"C"	PASS	The test case was derived to check to see if the expected output which is the character, "C", to the actual output of the variable p. The variable p stores the new note.	It was implemented by using String called expected where it stored the expected character. The Pitch class is getting called by the variable p where it is storing a new note. The

					expected variable and the p.getStep() is then being compared by using the function assertEquals.
testAlter_Normal	0	0	PASS	The test case was derived to check to see if the expected output which is the numeric value ,0, to the actual output of the variable p. The variable p stores the new note.	It was implemented by using int to store the expected value in this variable. The Pitch class is getting called by the variable p where it is storing a new note. The expected variable and the p.getAlter() is then being compared by using the function assertEquals.
testAlter_Sharp	1	1	PASS	The test case was derived to check to see if the expected output which is the numeric value ,0, to the actual output of the variable p. The variable p stores the new note.	It was implemented by using int to store the expected value in this variable. The Pitch class is getting called by the variable p where it is storing a new note. The expected

					variable and the p.getAlter() is then being compared by using the function assertEquals.
testOctave_Normal	5	5	PASS	The test case was derived to check to see if the expected output which is the numeric value ,5, to the actual output of the variable p. The variable p stores the new note.	It was implemented by using int to store the expected value in this variable. The Pitch class is getting called by the variable p where it is storing a new note. The expected variable and the p.getOctaver() is then being compared by using the function assertEquals.
testOctave_Sharp	5	5	PASS	The test case was derived to check to see if the expected output which is the numeric value ,5, to the actual output of the variable p. The variable p stores the new note.	It was implemented by using int to store the expected value in this variable. The Pitch class is getting called by the variable p where it is storing a new note. The expected variable and the

					p.getOctave() is then being compared by using the function assertEquals.
testEquals	True False	True False	PASS	To check the equality between two pitches.	Creating two pitch objects with the same/different step, alter, and octave. Checks the equality by calling the equals method.

Why is it sufficient?

These test cases are sufficient to use because when writing out methods we wanted to check if the calculations for the alter, step, and octave are correct. These methods are important in terms of the MusicXML file output to get the correct values or characters when inputting a text file.

3.2 TestTabReader.java Class

TestTabReader.java					
Test Case Names	Expected	Actual	Results (Pass/Fail)	Why/How it was derived?	How it was implemented?
testReadFile	Reads the contents of test_tabs_reading.txt	Reads the contents of test_tabs_reading.txt	PASS	We used a standard text tab.	Each line in the file was compared to the expected list of strings.
testSplitMeasure	Returns a list of measure as a list of strings.	Returns a list of measure as a list of strings.	PASS	We wanted to split the measure wherever there is a line of vertical bars.	The measures returned by splitMeasure is compared against the expected measures.

testLineHasTabs	Returns true when the line contains at least 2 vertical bars and 2 dashes.	Returns true when the line contains at least 2 vertical bars and 2 dashes.	PASS	Testing to check if a string has 2 vertical bars and 2 dashes.	1.Returns true if the string has 2 vertical bars and 2 dashes 2. Returns false if the string has 1 vertical bar and 2 dashes 3. Returns false if the string has 2 vertical bars and 1 dash
testGetTuning	[E,B,G,D,A,E]	[E,B,G,D,A,E]	PASS	Testing standard tuning.	Adding the standard tuning to a list and comparing it to getTuning().
testGetTitle	Gets the file's name, test_tabs_reading	Gets the file's name, test_tabs_reading	PASS	To get the title of the text file.	The value returned by getTitle to test_tabs_reading
testGetInstrument	Returns the instrument whether its "Classical Guitar" or "Drumset".	Returns the instrument whether its "Classical Guitar" or "Drumset".	PASS	Reading two different text tabs, one for guitar and one for drums.	The value returned by getInstrument to the designated text file's instrument.
testLastCharacter	True	True	PASS	Checks to see if TabReader will parse a note that is right beside the final absolute symbol (last character in the line)	Implements to see of the expected note is beside the vertical bar.
testLastCharacterTechnique	True	True	PASS	To see if the expected	If the first character of the

				character of the measure equals the last note of the previous measure as a beginning value.	next measure is a technique (letter), then it makes sure that the last note from the previous measure is assigned that technique's "start" value Ex: |----------0|---p0----------| The 0 in the first measure will be pull-off start.
testGuitarTechniques	True	True	PASS	Checks if correct note attributes are assigned for specific notation (r – release, b – bend, p – pull-off start or pull off end, h – hammer-on start or hammer-on end)	Implemented by checking if the actual array and the expected array have specific notation from the note attributes which are then compared by techStart and texhStop.
testDrumBeams	True	True	PASS	Makes sure that the correct number of beams are assigned for drum Notes (1 beam for 8^{th} notes, 2 beams for 16^{th} notes, 3 beams for 32^{nd} notes)	Implemented by getting the notes of the drum instrument and checking to see if it equals to the number of beams.

Why is it sufficient?

These test cases are sufficient because it checks if the method is producing the right output or that it ensures that the method we have written is being used as the user expected. For example, the test case of the testReadFile checks whether or not an input text file can be read from software system.

3.3 TestNote.java Class

TestNote.java					
Test Case Names	Expected	Actual	Results (Pass/Fail)	Why/How it was derived?	How it was implemented?
testCompareTo_Negative	True	True	PASS	True when this note precedes the other note when being compared.	This was compared by using assertTrue to check if n1 and n2 is less than 0.
testCompareTo_Positive	True	True	PASS	True when the other note precedes this note when being compared.	This was compared by using assertTrue to check if n1 and n2 is greater than 0.
testCompareTo_Equals	True	True	PASS	True when this note is either above or below the other note.	This was compared by using assertTrue to check if n1 and n2 is equal to 0.
testDrumGhost	True	True	PASS	Checks if correct note attributes are assigned for specific notation (g – ghost)	Creates an object and calling the note class. Storing the new object note in the variable called note and returning true if the note

					attributes are assigned to a specific notation.
testDrumRoll	True	True	PASS	Checks if correct note attributes are assigned for specific notation (b – roll)	Creates an object and calling the note class. Storing the new object note in the variable called note and returning true if the note attributes are assigned to a specific notation.
testDrumFlam	True	True	PASS	Checks if correct note attributes are assigned for specific notation (f – flam)	Creates an object and calling the note class. Storing the new object note in the variable called note and returning true if the note attributes are assigned to a specific notation.
testDrumDrag	True	True	PASS	Checks if correct note attributes are assigned for specific notation (d – drag)	Creates an object and calling the note class. Storing the new object note in the variable called note and returning true if the note attributes are assigned to a specific notation.

testHiHatRoll	True	True	PASS	Checks if correct note attributes are assigned for specific notation (B – roll for hi hat)	Creates an object and calling the note class. Storing the new object note in the variable called note and returning true if the note attributes are assigned to a specific notation.
testDrumAccent	True	True	PASS	Checks if correct note attributes are assigned for specific notation (O – accent for snare)	Creates an object and calling the note class. Storing the new object note in the variable called note and returning true if the note attributes are assigned to a specific notation.
testGetPitch	True	True	PASS	Checks to see if the two objects are equal to each other.	Creates two objects, one of the actual and one of the expected by calling the Pitch class and returns true if the actual object and the expected are equal to each other.

Why is it sufficient?

These test cases are sufficient to use because in the class note there are codes written for the guitar/bass in terms of notes. There are also codes written for the drum instrument in terms of notes. (Still in process of completing). These codes need to ensure that it is reading the note of the desired instrument accordingly. If a note is not being expected to the user needs (which would be us) it can help determine where the problem is. For example, it can help the user to check if it is giving the desired tuning or fret number and so on.

3.4 TestMeasure.java Class

TestMeasure.java					
Test Case Names	Expected	Actual	Results (Pass/Fail)	Why/How it was derived?	How it was implemented?
testMeasure	Constructor successful	Constructor successful	PASS	To check whether the constructor does not crash.	This was implemented using the try and catch block.
testSortArray	Sorts the notes by non-descending order.t	Sorts the notes by non-descending order.	PASS	This was derived by testing to see if the measures are in ascending order.	This was implemented by taking a text file and looping through the measures to put them in ascending order.

Why is it sufficient?

These test cases are sufficient to our software system because it checks whether the measures are in ascending order in order to produce the designated MusicXML file. This is sufficient to the software system as to it determines the order of the measures and how it would be disignated on the designated XML file.

3.5 TestAttributes.java Class

TestAttributes.java					
Test Case Names	Expected	Actual	Results (Pass/Fail)	Why/How it was derived?	How it was implemented?
testAttributes	Constructor successful	Constructor successful	PASS	To check whether the constructor	It was implemented using the try and catch block.

				does not crash.	

Why is it sufficient?

These test cases above are sufficient to our software system because in a guitar tab or bass tablature there are different tuning. With this said, the different tuning defines the different pitches. However, it is crucial that the testAttributes count for all of this. As a result of this, coding these test cases aids in the process to check whether the guitar tuning has been done properly or if there are any error that as a user need to take care off.

3.6 TestUnpitch.java Class

Unpitch.java					
Test Case Names	Expected	Actual	Results (Pass/Fail)	Why/How it was derived?	How it was implemented?
test	Step: A Octave: 5	Step: A Octave: 5	Pass	The test case was derived to check to see if the expected output returns the according steo and octave of a drum note.	It was implemented by creating an object to store the expected value in this variable. The Unpitch class is getting called by the variable up where it is storing the object. The expected variable is used to print to see if that drum set component produces the right octaves and step.

Why is it sufficient?

The test cases are sufficient for the Unpitch class because it determines whether the methods in the Unpitch class are correct in terms of producing the correct steps and octaves. These methods are important in terms of producing the correct MusicXML file and having the right values for these variables.

4.0 Test Case Description Front End

4.1 ApplicationTest.java Class

ApplicationTest.java					
Test Cases Names	Expected	Actual	Pass/Fail	Why/How it was derived?	How it was implemented ?
Start	Loads The PrimaryStage FXML Stage	Loads the FXML Stage	Pass	Implemented to add the stage for testing	The primaryStage will be injected into the test runner.
should_contain_button _with_convert	Button: convert Label: Convert	Button: convert Label: Convert	Pass	Implemented to verify that the convert button is its appropriate box and loaded.	Using the FXRobot, the labels were buttons were matched with the expected string.
should_contain_button _start	Button: startButton Label: Start	Button: startButton Label: Start	Pass	Implemented to verify that the start button is its appropriate box .	Using the FXRobot, the labels were buttons were matched with the expected string.
should_contain_button _with_help	Button: helpButton Label: Help	Button: helpButton Label: Help	Pass	Implemented to verify that the help button is its appropriate box .	Using the FXRobot, the labels were buttons were matched with the expected string.
should_contain_button _with_save	Button: save Label: Save MusicXML File	Button: save Label: Save MusicXML File	Pass	Implemented to verify that the save button is its appropriate box .	Using the FXRobot, the labels were buttons were matched with the expected string.

should_contain_button _with_select	Button: select Label: Select File	Button: select Label: Select File	Pass	Implemented to verify that the select button is its appropriate box .	Using the FXRobot, the labels were buttons were matched with the expected string.
should_contain_button _with_feature	Button: featureButton Label: Select Time Signiture	Button: featureButton Label: Select Time Signiture	Pass	Implemented to verify that the feature button is its appropriate box .	Using the FXRobot, the labels were buttons were matched with the expected string.
should_contain_button _with_timSigButton	Button: titleButton Label: Specify Title	Button: titleButton Label: Specify Title	Pass	Implemented to verify that the title button is its appropriate box .	Using the FXRobot, the labels were buttons were matched with the expected string.
should_contain_button _with_composerButton	Button: composerButton Label: Specify Composer	Button: composerButton Label: Specify Composer	Pass	Implemented to verify that the composer button is its appropriate box .	Using the FXRobot, the labels were buttons were matched with the expected string.
emptyTextArea	TextArea: inputBox TextInputControlMatchers: (empty string)	TextArea: inputBox TextInputControlMatchers: (empty string)	Pass	Implemented to check whether or not after clicking start, the input area is empty for the user.	Using the FxRobot, the robot clicked the appropriate series of buttons to show that the input box is empty
showsErrorInput	TextArea: outputBox TextInputControlMatchers: (empty string)	TextArea: outputBox	Pass	Implemented to check if the program	Using the FxRobot, the robot clicked

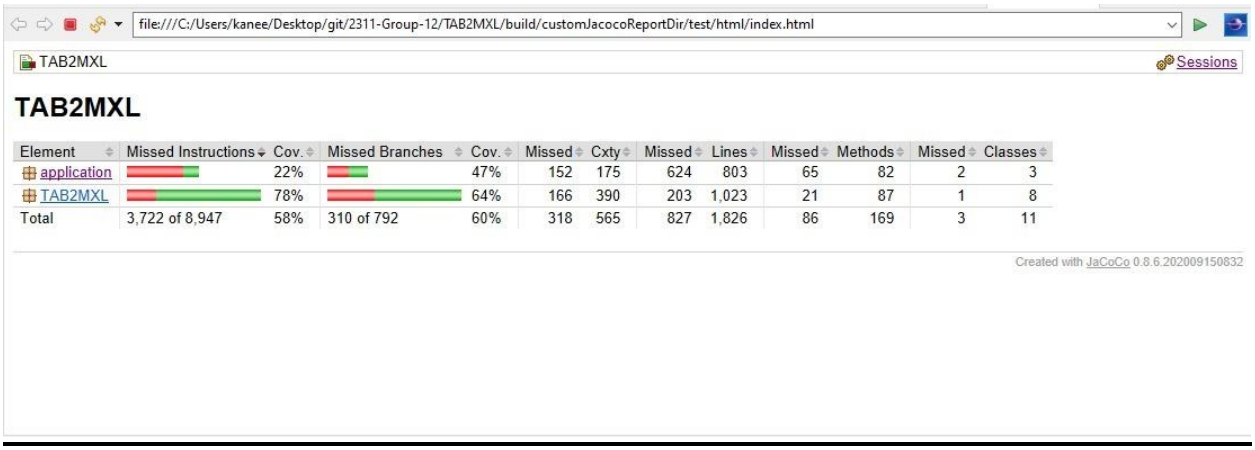
		TextInputControl Matchers: (empty string)		can gracefully inform the user that there is an error reading their program. When the program cannot read the file, output box is empty.	the appropriate series of buttons to show that the output box is empty
properInputTest	Label: InstrumentID LabeledMatchers: “No Instrument Detected”	Label: InstrumentID LabeledMatchers: “No Instrument Detected”	Pass	Implemented to check if the program can detect the instrument and inform the user which instrument was detected.	Using the FxRobot, the robot clicked the appropriate series of buttons to show that the instrumentID label is updated.
verifySaving	Button: save Label: Save MusicXML File	Button: save Label: Save MusicXML File	Pass	Implemented to mimic the user saving a file to their desktop. This is derived to check if the software can save the input	Using the FxRobot, the robot clicked the appropriate series of buttons to show that once the user saves, the user returns to main screen where the save button is.
testSaveTestFile	Button: start Label: Start	Button: start Label: Start	Pass	Implemented to mimic the user saving their	Using the FxRobot, the robot clicked the

				text file that they changed. The user can save the file to their desktop.	appropriate series of buttons to show that once the user saves the text file, the user is returned back to the primary stage.

Why is it sufficient?

The test cases are sufficient for the ApllicationTest.java class is because it guarantees that the different features interact with the back end of the software system. These test cases ensures that the operation of the features are working accordingly and the functionality GUI is correct. This is essential for front end testing because it demonstrates that the software application created for the purpose of translating the according XML file.

5.0 Coverage Metrics



The coverage metrics shown above demonstrates the test coverage in two packages. The package called application is made for the front end and test features on the GUI. The package called TAB2XML demonstrates the test coverage in regards of the backend of the software system. In terms of backend, testing the different methods tested reach a coverage of 76% which is good for our project. This means that out of the all the methods created in the backend 76% of them were tested. The package called application demonstrates the test coverage in regards of the frontend of the software system. The coverage for frontend is 22%, this is because with frontend testing of the GUI there isn't a lot of testing to see if methods produce the right outcome compared to the backend. As a result, testing of the front end includes features to see if the functionality of the buttons works accordingly and if the labels are correct.

6.0 Testing Requirement and Features

The testing requirements for each test case is using the JUnit 5 library that needs to be installed in order to run the test cases. This is an essential part of the written test cases because without it the test cases will not run. When writing the test case for our methods, the proper syntax at the beginning is to write *@Test* to ensure that the software will know that this is a test case method. Test cases are implemented for all three instrument, guitar, drum and bass. These test cases are implemented to check if the backend code produces the designated output in order to produce the correct MusicXML file.

The primary focus of the GUI testing was to ensure that all components that user would interact with were loaded and had the appropriate labels. In our TAB2Reader software, the front end testing checks whether or not the main functions of the program that is selecting a file, converting a file and saving a file works completely perfect for the user. Other smaller features that the user may select was dealt in the backend. For the GUI testing, various libraries were used including TestFX and other libraries such as Hamcrest and FxRobot, a robot that is a robot mimicking user interaction. Furthermore, the testing class was created by extending the Application class.