TESTING DOCUMENT

Project: TAB2XML
Course: Software Development Project (EECS 2311)

THE TEAM (GROUP 12):

Matteo Pulcini - [217536756]
Krishna Raju - [218199497]
Patrick Qi - [218095091]
Riffi Manoj - [218061986]

Table Of Contents

1.0 Introduction

Every piece of software should have tests to verify its correctness. In this application they're a number of different features that rely on each other and in order to verify that each piece is functioning correctly, tests must be created.

1.1 Disclaimer

This system was designed using JavaFX. The tablature and sheet music was drawn on the screen using the pane object. Oftentime through the manual, you may see x-location or y-location this refers to the x and y of a cartesian plane as the guitar tablature and sheet music for the drums are drawn on with reference to a cartesian plane.

2.0 Tests

2.1 TestingMeasure.java

This class is intended to make sure all the measures are being added correctly for the guitar tablature and for the drum measures as they are not the same. They're multiple areas that are crucial to check and an error would be detrimental for the whole output.

2.1.1 Guitar & Drum Measure

Test 1: `one_measure_guitar() && one_measure_drums()`
Context: This test case was derived to make sure a measure could be inserted on the pane in a correct location. This is important because one measure sets the reference for other measures and the first one should be inserted in the correct location to allow for follow up measures.
Implementation: A custom measure object is created and drawn on. While this is occurring a counter triggers and sets the current measure count to one. At the end of the function responsible for drawing, comparing the counter to the number of added measures.
Pass/Fail Conditions: This test passes if the counter indicates one measure has been printed on the screen and fails if more than one have been counted.

Test 2: `multiple_measures_guitar() && multiple_measures_drums()`
Context: This test case was derived to make sure multiple measures after the first can be inserted on a pane. This is important because multiple measures should not overlap, and should start where the last was left off.
Implementation: A custom measure object is created and we draw 20 measures on it. Whenever a measure is drawn on a Pane(Java Object) using the pane.getChildren.add method, a counter triggers to record the *currentMeasureCount* which is compared in the JUnit test.
Pass/Fail Conditions: This test passes if the counter indicates multiple measures have been counted and fails in the expected result differs from the actual result of counted measures.

Test 3: `four_bars_guitar_measure() && three_bars_drums_measure()`
Context: This test case was derived to take sure a guitar measure has six horizontal bars on the measure. This is important to check because this is consistent throughout the program and an error could propagate forward making the output unreadable if left unchecked.
Implementation: A constructor creates the custom measure object and the number of lines is passed in said constructor. This test goes to the class and verifies the function creating the lines repeats as many times as the constructors input.
Pass/Fail Conditions: This test passes if the amount of lines drawn in between measures is counted correctly four inner bars for guitar and three inner bars for a drums measure, if the counter differs from these values the test will fail.

2.1.2 Multiple Measures

Testing multiple measures allows one to check to see if the horizontal position of each measure is adjusted after the width of the measure times the amount of measures is larger than the GUI.

Test 4: `multiple_measures_on_newline()`
Context: Tests to make sure that measures can be inserted at different horizontal locations when they would otherwise be out of view for the user. This is important because the GUI has a maximum width, so in order to see long outputs music must be inserted in a scroll pane.
Implementation: This is created by recording the measure count divided by measures per line and then by multiplying the length the measure moves down in the y-direction. This result can be compared to the function *getCurrentTopOfMeasureHeight(),* which records the y coordinate of the top of the measure.
Pass/Fail Conditions: This test passes if the variable recording the y-position of new measures changes every time the measure width exceeds the length of the page and it fails if the expected value differents from the actual value. E.x. The eighth measure should start at 600 units down.

2.1.3 Adding T_A_B on each new Guitar line

Test 5: `multiple_tab_text_for_guitar_measure() && tab_text_for_guitar_measure()`
Context: This tests when a new line is created (a line with more than three measures), we need to add the letters T, A, and B. This is important to make sure the text appears only on one measure per new line.
Implementation: Everytime a new line is created a counter is triggered and a method should be run to add the letters on the pane, this is verified by comparing the counter with the amount of *(measures/(measure per line))+ 1.*
Pass/Fail Conditions: This test passes if the amount of times T_A_B are drawn onto the screen equals the amount of new lines where a measure starts, it fails if this is false.

2.1.3 Adding || on each new Drum line

Test 6: `single_doubleLines_for_drum_measure() && multiple_doubleLine_for_drum_measure()`
Context: This tests when a new line is created (a line with more than three measures), we need to add a || at the beginning of every newline. This is important to make sure the text appears only on one measure per new line.
Implementation: Everytime a new line is created a counter is triggered and a method should be run to add the letters on the pane, this is verified by comparing the counter with the amount of *(measures/(measure per line))+ 1.*
Pass/Fail Conditions: This test passes if the amount of times ‘I I’ are drawn onto the screen equals the amount of new lines where a measure starts, it fails if this is false.

2.2 TestingGuitar.java

This class is intended to make sure all the guitar notes are being added correctly. They're multiple areas that are crucial to check and an error would be detrimental for the whole program.

2.2.1 Guitar Notes

Test 1: `correctly_added_one_note() && correctly_added_many_notes()`
Context: Tests to make sure the position of the first note and multiple notes in the measure is correct, this is important because if the first note is added in the wrong location, all notes will be affected and if multiple notes are inserted incorrectly, the outlay will be unreadable or incorrect.
Implementation: Everytime a new note was drawn on the screen a counter would add one. This test case adds an arbitrary number of notes and checks to see if the counter resembles this arbitrary number.
Pass/Fail Conditions: This test passes if the correct amount of notes are drawn onto the screen for example, if the input contains six notes, we should expect six notes in the output if this is false the test fails.

2.2.2 Guitar Note Position x-axis

Test 3: `note_X_position() && note_X_reset_beginning_measure()`
Context: Tests to make sure that the x-position of multiple notes in a cartesian plane are correct and to make sure when a note is always placed at the beginning of a new measure. This is important because if the note in the x-position is not reset or added correctly, the entire output can be altered.
Implementation: Everytime a new note is imputed on the screen its position in the x-axis is determined by how many notes are in a measure. We predict value and compare it to the output of our function to make sure it is behaving unexpectedly for eighth notes.
Pass/Fail Conditions: This test passes if the x-position of the note is correct. For example, if notes expected position is equal to the actual position depending on the duration then the test will pass, if not the test will fail.

2.2.3 Guitar Note Position y-axis

Test 4: `note_Y_move_down_new_measure() && starting_note_Y_position_all_string()`
Context: Tests to make sure that the y-position of multiple notes in a cartesian plane are correct and to make sure when a measure is moved downwards with respect to the y-axis on the pane, the y-position for the note follows. This is important because if the y-position of the note does not move down with the measure, the output will be unreadable.
Implementation: Each string has a specific y-axis location.
Pass/Fail Conditions: This test passes if the y-position of the note is correct. For example, if notes expected position is equal to the actual position depending on the string and fret then the test will pass, if not the test will fail.

2.2.4 Guitar Slurs

Test 5: `test_slurs_input_guitar() && test_slurs_output_guitar()`
Context: Tests to make sure that correct amount of slurs are drawn on to the screen and that they are in the correct positions.
Implementation: This is determined through a method which calculates the location of the slur based on the X-position of two notes. Every time a slur is drawn on the screen a counter is activated and that counter is checked in the test cases. Also the method which draws the slurs is testing by comparing the expected output to the actual output.
Pass/Fail Conditions: If the amount of slurs that appear in the input are equal to the amount of slurs that appear in the output with the correct positioning the test will pass, if not the test will fail.

2.2.5 Guitar Slides

Test 6: `test_slides_input_guitar() && test_slides_output_guitar()`
Context: Tests to make sure that correct amount of slides are drawn on to the screen.
Implementation: An object is created which contains all the information about an individual note, including if it is a slide. If a certain note has its ‘slide’ flag set to true, a counter is increment and compared to the expected value in several jUnit tests.
Pass/Fail Conditions: If the amount of slides that appear in the input are equal to the amount of slides that appear in the output with the correct positioning the test will pass, if not the test will fail.

2.2.6 Guitar Grace Notes

Test 6: `test_grace_guitar()`
Context: Tests to make sure that correct amount of grace are drawn on to the screen and that the grace notes are being passed from the parser to each indivdiual object properly so the 'grace flag' is set to true and the note is drawn at a smaller size.
Implementation: The notes individually added with some being grace notes, the constructor sets which notes are grace and after the parser is run a method is run on each elemented in the result from the parser to see if the expected notes are grace notes.
Pass/Fail Conditions: If the amount of grace notes that appear in the input are equal to the amount of grace notes that appear in the output the test will pass, if not the test will fail.

2.2.7 Guitar Note Duration Bars

Test 7: `test_bar_16_note() && test_bar_eight_note() && test_bar_x_location()`
Context: Tests to make sure the bars under the guitar measures represent the correct duration based on the notes above.
Implementation: The formatting of the bars which is determined by the output is compared to the input and if the bar matches what 16th or 8th notes or whole notes.
Pass/Fail Conditions: If the expected value used to create the duration bar equals the actual value used to draw the duration bar under the measure.

2.3 TestingDrum.java

This class is very similar to the TestingGuitar class, as the positioning of the notes in the x-axis and y-axis work in similar ways.

2.3.1 Drum Notes

Test 1: `correct__added_one_notes() && correctly_added_many_notes()`
Context: Tests to make sure the position of the first note and multiple notes in the measure is correct in a cartesian plane. This is important for the same reasons as the guitar notes.
Implementation: Same as guitar implementation but for drum notes.
Pass/Fail Conditions: Same as guitar implementation but for drum notes.

2.3.2 Drum Note Position x-axis

Test 2: `note_X_position() && note_X_reset_beginning_measure()`
Context: Tests to make sure that the x-position of multiple notes in a cartesian plane are correct and to make sure when a note is always placed at the beginning of a new measure. This is important for the same reasons as the guitar notes.
Implementation: Same as guitar implementation but for drum notes.
Pass/Fail Conditions: Same as guitar implementation but for drum notes.

2.3.3 Drum Note Position y-axis

Test 5: `note_Y_move_down_new_measure() && starting_note_Y_position_all_string()`
Context: Tests to make sure that the y-position of multiple notes in a cartesian plane are correct for multiple measures when they are inserted on a new line. This is important for the same reasons as the guitar notes.
Implementation: Same as guitar implementation but for drum notes
Pass/Fail Conditions: Same as guitar implementation but for drum notes.

2.3.4 Drum Slurs

Test 6: `test_slurs_input_drums() && test_slurs_output_drums()`
Context: Tests to make sure that correct amount of slurs are drawn on to the screen and that they are in the correct positions.
Implementation: This is determined the same way as the guitar slurs.
Pass/Fail Conditions: Same as guitar implementation but for drum notes.

2.3.5 Drum Grace Notes

Test 7: `test_grace_drums()`
Context: Tests to make sure that correct amount of grace are drawn on to the screen and that the grace notes are being passed from the parser to each indivdiual object properly so the ‘grace flag’ is set to true and the note is drawn at a smaller size.
Implementation: This is determined the same way as the drum grace notes.
Pass/Fail Conditions: Same as guitar implementation but for drum notes.

2.3.6 Drum Test Repeats

Test 7: `testNumberOfRepeats()`
Context: Tests to make sure the correct amount of repeats are calculated for specific measures.
Implementation: A sample measure is created which is expected to repeat a certain number of times, the expected repeat output is compared against the actual result to see if the methods are performing correctly.
Pass/Fail Conditions: This test passes if the number input number of repeats equals the value in the output (expected equals actual) and it fails if this is false.

2.4 MusicPlayerTest.java

Test 1: `testPlayerController()`
Context: `PlayerController.java` is supposed to be a controller for the music player GUI, not to be used as an individual class.
Implementation: When the class gets instantiated by itself, (not via FXMLoader) an exception is thrown due to the GUI component variables not being injected. The test fails if no exception is thrown.

Test 2: `testBasicDrumSequence()`
Context: To ensure basic functionality of generating a drum sequence.
Implementation: Test would pass if generated sequence should match the correct verified sequence.

Test 3: `testGuitarDefaultSignature()`
Context: The implementation has an edge case when no time signature is specified. (it would default to something much slower) This tests if that is properly handled.
Implementation: The test ensures that the generated guitar sequence is playing with a time signature of 4/4 (default), by comparing it with a verified sequence.

Test 4: `testUndefinedInstrument()`
Context: Since the program itself supports only 13 different percussion sounds, need a test when the user specifies an instrument ID that is not supported.
Implementation: The test tries to obtain an instrument mapping with the non-existent instrument ID, and fails.

Test 5: `testTempoFactor()`
Context: This test is to check if the tempo played back matches the tempo set by the user.
Implementation: The way the speed of the playback is scaled is using a `tempoFactor`. The test checks if userTempo = original tempo * tempoFactor.