TAB2XML Testing Document

For Version 0.4.0-alpha.1

2021 March 29

Contents

1 Front-End Tests

Many of the front-end tests are implemented using a `ViewBot`, a class that simulates a GUI.

1.1 PresenterTest

These test cases were derived by making one test for each important method of `Presenter`.

- `testConvert()`: Tests that the Presenter's convert method correctly interacts with the input (gets input text & instrument, uses the parser to convert the tab and sets the output correctly). It does not check that the converted MusicXML is correct - that is the backend tests' job.
  It is implemented by first loading a sample input from a file and converting it using the parser to get the expected output. Then, the ViewBot simulates inputting this text and selecting the correct instrument (guitar). Then, the presenter's convert() method is called. Finally, the ViewBot's output text is compared with the expected output.

- `testConvertAndSave()`: Tests that the Presenter's `convertAndSave()` method works properly. It is basically a combination of `testConvert` and `testSaveToFile`. In addition, it tests that the boolean argument of `convertAndSave` works properly by running twice, once per argument.
  It is implemented by loading sample input and issuing a convertAndSave command, similarly to `testConvert`. Then, both the View's output and the test file used as output have their contents checked with the correct text. Like in `testConvert`, this output is generated by the backend code.

- `testErrorNoSelectedFile()`: Tests that the system's commands properly fail (without halting) when the user does not select a file. Also ensures that the system does not attempt to read from or write to any files in this situation.
  It is implemented by calling the Presenter's methods before setting any

file. A simple guitar tab is put into the View's input to prevent any errors caused by the empty input. Variables counting the number of attempts to read from and write to files is implemented in the Presenter to ensure none of these operations occur.

- `testErrors()`: Tests that calling certain operations with an illegal state of the View correctly throws errors.
  It is implemented by using the ViewBot's methods to put it in the illegal state, then executing the Presenter's methods inside a assertThrows lambda.

- `testLoadFromFile()`: Tests that the "Load from File" command works properly.
  It is implemented by using sample text in a file. A presenter uses its `loadFromFile()` command to load the text into a ViewBot, and the ViewBot's input text is compared with the text that was in the file.

- `testSaveToFile()`: Tests that the "Save to File" command works properly.
  Some text is put in the output of a ViewBot. Then, a Presenter uses its `saveToFile()` command to save the text to a file. The file's text is compared with the text that was inputted in the ViewBot.

This testing is sufficient because every method of the Presenter is covered by a test (except the constructor, which is trivial and has only one line of code). The Presenter's methods are simple enough that only one test is necessary for each.

1.2 PromptingTextAreaTest

These test cases were derived by making one test case for each of the important functionalities of the `PromptingTextArea`: the colour and font of the prompt, the prompt text disappearing when the text box is focused, typing text in the box, and setting the area's font.

- `testAutoPromptFont()`: Tests that the PromptingTextArea correctly auto-creates prompting and non-prompting fonts. It is implemented by setting the area's prompt font to null and then calling getPrompt-Font(). It expects that a newly created font is returned.

- `testPromptColourFont()`: Tests that the text box's colour and font is set properly.
  This test works by creating a PromptingTextArea, and setting custom fonts with `setRegularFont()` and `setPromptFont()`. The prompt is disabled, and the active colour and font is checked for correctness. The prompt is enabled, and the same checks are performed.

- `testPromptFocusChanges()`: Tests that the text box reacts properly to focus changes.
  This test is implemented by simulating gaining and losing focus on the text box, and testing that the text box's text updates correctly.

- `testPromptTyping()`: Tests that the text box reacts properly to typing and when methods are run on it (setText and setPromptText).
  This test works by undergoing several operations (adding and deleting text, gaining and losing focus, changing the prompt text, manually enabling or disabling the prompt) while checking the text in the box is correct after each step. Typing and deleting text is simulated using the setText method.

- `testSetFont()`: Tests that fonts are set correctly.
  This test is implemented by disabling the prompt, and setting the font. The regular, prompt and active fonts are checked for correctness. Then, the prompt is enabled and the regular, prompt and active fonts are checked again. The test is repeated, but the prompt starts enabled and is switched to disabled in the second step. This is done because the setFont method behaves differently based on whether the prompt is enabled or disabled.

This testing is sufficient because every method of the PromptingTextArea is tested at least once, and all important or complex methods have tests dedicated to them and their related methods: `testPromptColourFont` tests

setRegularFont() and setPromptFont(); `testPromptTyping` tests setText() and setPromptText(); `testSetFont` tests setFont(). All other public methods are simple getters or setters, or methods that trivially call one of the tested methods. In addition, the gain or loss of focus, an important feature of the `PromptingTextArea`, has its own dedicated testing method.

1.3 TimeSignatureTest

This is a small test that tests the time signature setting functionality. It was derived by thinking of situations that could cause problems, and creating one test for simple, non-problematic situations. Both tests are implemented by creating an `XMLMetadata` instance with specific time signatures, and checking the contents of the maps returned by `getTimeSignatures()` and `getTimeSignatureRanges()`.

- `testTimeSignatures()`: Tests a set of time signatures where the measure ranges do not overlap.

- `testNonDisjointIntervals()`: Tests a set of time signatures where the measure ranges do overlap.

This is sufficient testing because it tests an example of every major scenario, and test data for `XMLMetadata` (which has functionality other than the time signatures being tested, though this functionality is simple enough that it does not need to be tested) is 93 %. All instructions in `XMLMetadata` that relate to time signatures are covered by this test.

1.4 ViewTest

These tests were derived by making one test for each major method of the View interface.
All tests in this section are run once per View supported by the program, and once for the ViewBot. This ensures that all of the Views support every possible feature. Any test in this section that requires use of an unimplemented optional method is skipped.
All three of these tests are implemented by setting the paramater to some value, then comparing the value set to the value returned by the appropriate

get method.

- `testInputText()`: Tests that all of the standard views can correctly get and set their input text.

- `testOutputText()`: Tests that all of the standard views can correctly get and set their output text

- `testInstrumentSelection()`: Tests that all of the standard views can correctly get and set their instrument selection

This testing is sufficient because, like in the Presenter, every important method in the View interface is tested by one test, except `showErrorMessage(String, String)`. The showErrorMessage method cannot be tested automatically (because I do not want to specify **how** an error message is shown, only that one is shown), and it is trivial enough that I am not worried about it breaking (As of the time this document was written, all implementations of this method have only one line of code). The View's methods are also simple enough that only one test per View is needed for each method.

2 Back-End Tests

2.1 ParserTest

These tests were derived by making sure that the parser was correctly interpreting the information provided through a text tab.

- `testScore()`: Tests that there is the correct amount of Staffs within a given Score.
  This test was created by creating a string of a sample text tab, and using the Parse Tree to locate and count all of the staffs contained within the score, then comparing it with the expected amount of staffs.

- `testStaff()`: Tests that there is the correct amount of Measures and number of strings in a given Staff.

This test was created by creating a string of a sample text tab of a single staff, and using the Parse Tree to locate and count every measure and every string, then comparing those with the number of expected measures and strings.

- `testTuning()`: Tests that each string in a staff is the expected tuning.

This test was created by creating a string of a sample text tab of a single staff, and using the Parse Tree to locate the tuning of each string and compare that with the expected tunings of each string. For inputs that do not contain string tunings, the expected tuning is the default guitar tuning.

- `stringItemCompareTo()`: Tests that the parser reads the notes in the correct order that they appear in the text tab.

This test was created by hard coding an array of different notes, with different positions in the tab, and adding them to an array, then comparing each note in the array to an array of each note in the order they are expected.

This testing is sufficient because there are tests for each basic component of a text tab (for example, measures or strings), and ensures that the parser is able to accurately interpret and store the information. Since the prototype is expected to handle simple tabs, only testing for the simplest components of a text tab were created. In the future this tester will have testing for more complex components, and testing for different components that the parser is not yet set up to interpret.

2.2 NoteTest

These tests were derived to make sure that note objects, which contain valuable information about notes that can be used in the xml conversion process, can be properly created.

- `noteTest()`: Tests that notes have the correct name and index.

This test was created by passing note to test, expected name of note and expected index of note as the parameter.

- `testToNote()`: There are 2 versions of testTonote, and both of them have different arguments. The first tests the toNote method in the Note class and checks if a valid note is correctly converted, and the other one tests the invalid notes.

There are 2 testTonote. The first was created by passing the string input(*"tune + fret"*) and the string this note is on, and checks if this was a valid note and if it was converted correctly by comparing it to an expected note. The 2nd one was created just by passing string input(*"tune + fret"*) and checks if an invalid note was entered by using exceptions.

This is sufficient testing because it checks that our system properly handles creating Note objects, which is a very important step in translating the information from text tabs to xml because notes are the main focus of learning songs through text tab. By testing the correctness and validity of these note objects, we can be sure that the notes that appear in a text tab will have the necessary information used in xml.

2.3 MeasureNarrowingTest

These were derived by considering the operations of `MeasureNarrowing` (including private methods) as well as the possible text tabs that could cause problems.
Each was implemented by loading a text tab from a file, then performing an operation on the loaded text tab, then checking the resulting tab against an output string. Some tests do this twice for more confidence.

- `testBottomRightCorner()`: Tests the `bottomRightCorner()` private method.

- `testDelinearize()`: Tests the `delinearize()` private method.

- `testExtractDecoratedMeasure()`: Tests the `extractMeasureRange` method with the Capricho Arabe tab (which has a lot of extra "decoration" around its measure text)

- `testExtractMeasure()`: Tests the `extractMeasureRange` method for a simple input (one measure at a time, one "row" of text tab)

- `testExtractMultilineMeasure()`: Tests the `extractMeasureRange` method for a complex input (tests a multi-row text tab, extracted range goes across a row boundary)

- `testExtractRepeatedMeasure()`: Tests the `extractMeasureRange` method on a tab with a repeated measure (since the method relies on the '|' character to delimit measures, repeated measures can cause errors by having two '|' characters).

- `testLinearize()`: Tests the `linearize()` private method.

- `testReplaceMeasure()`: Tests the `replaceMeasureRange` method for a simple input.

- `testReplaceMultilineMeasure()`: Tests the `replaceMeasureRange` method for a complex input.

- `testTopLeftCorner()`: Tests the `topLeftCorner()` private method.

This is sufficent testing because multiple distinct tabs are tested, and the code coverage for `MeasureNarrowing` is 98 %. Its package-private static member class `StringPosition` has 69 % code coverage, but the uncovered methods are all also unused (and all of them are trivial or autogenerated by Eclipse).