TAB2XML Testing Document

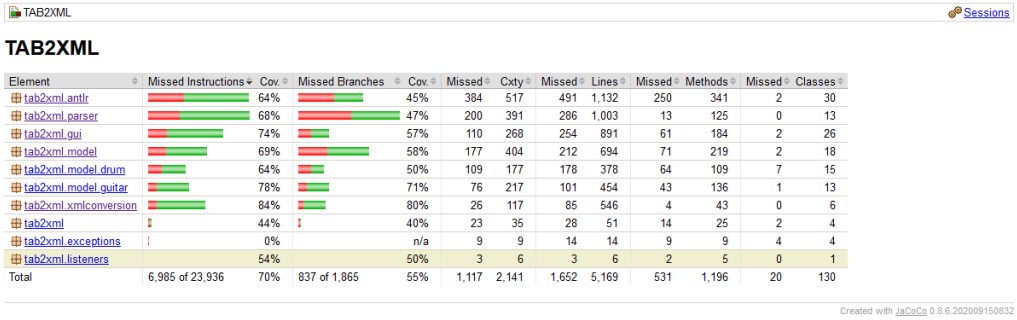
For Version 1.0.0

2021 April 13

Contents

1 Introduction

This document details the testing used in TAB2XML, how they were derived, how they are implemented, and why the testing is sufficient.



TAB2XML — Sessions

TAB2XML

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
tab2xml.antlr		64%		45%	384	517	491	1,132	250	341	2	30
tab2xml.parser		68%		47%	200	391	286	1,003	13	125	0	13
tab2xml.gui		74%		57%	110	268	254	891	61	184	2	26
tab2xml.model		69%		58%	177	404	212	694	71	219	2	18
tab2xml.model.drum		64%		50%	109	177	178	378	64	109	7	15
tab2xml.model.guitar		78%		71%	76	217	101	454	43	136	1	13
tab2xml.xmlconversion		84%		80%	26	117	85	546	4	43	0	6
tab2xml		44%		40%	23	35	28	51	14	25	2	4
tab2xml.exceptions		0%		n/a	9	9	14	14	9	9	4	4
tab2xml.listeners		54%		50%	3	6	3	6	2	5	0	1
Total	6,985 of 23,936	70%	837 of 1,865	55%	1,117	2,141	1,652	5,169	531	1,196	20	130

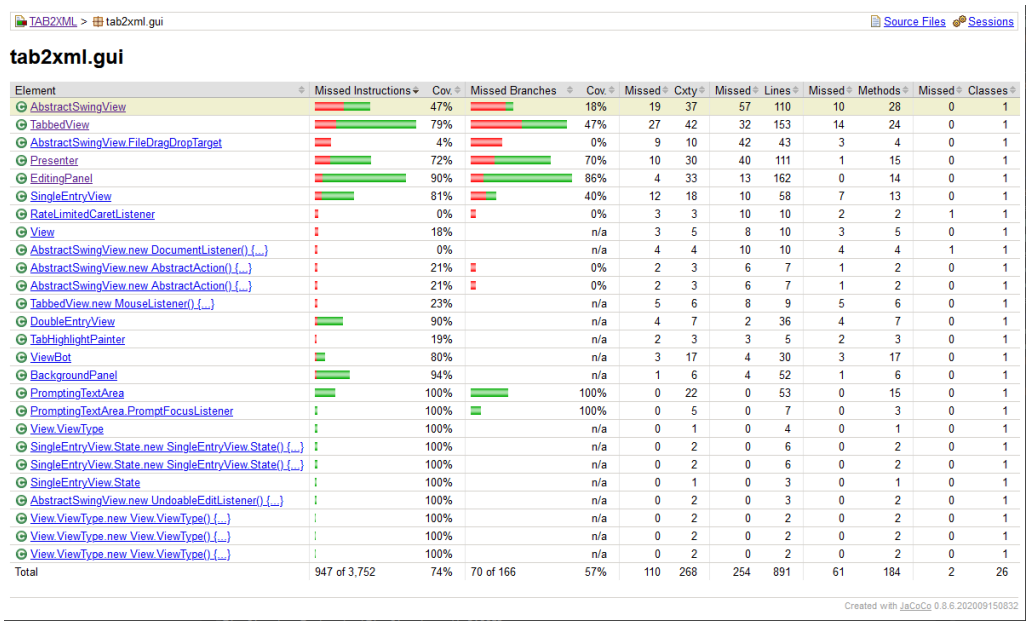
Created with JaCoCo 0.8.6.202009150832

Figure 1: Test coverage report for the whole project, as of 2021-04-13.

2 Front-End Tests

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

2.1 Code Coverage and Test Sufficiency



TAB2XML > tab2xml.gui — Source Files — Sessions

tab2xml.gui

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
AbstractSwingView		47%		18%	19	37	57	110	10	28	0	1
TabbedView		79%		47%	27	42	32	153	14	24	0	1
AbstractSwingView.FileDragDropTarget		4%		0%	9	10	42	43	3	4	0	1
Presenter		72%		70%	10	30	40	111	1	15	0	1
EditingPanel		90%		86%	4	33	13	162	0	14	0	1
SingleEntryView		81%		40%	12	18	10	58	7	13	0	1
RateLimitedCaretListener		0%		0%	3	3	10	10	2	2	1	1
View		18%		n/a	3	5	8	10	3	5	0	1
AbstractSwingView.new DocumentListener() {...}		0%		n/a	4	4	10	10	4	4	1	1
AbstractSwingView.new AbstractAction() {...}		21%		0%	2	3	6	7	1	2	0	1
AbstractSwingView.new AbstractAction() {...}		21%		0%	2	3	6	7	1	2	0	1
TabbedView.new MouseListener() {...}		23%		n/a	5	6	8	9	5	6	0	1
DoubleEntryView		90%		n/a	4	7	2	36	4	7	0	1
TabHighlightPainter		19%		n/a	2	3	3	5	2	3	0	1
ViewBot		80%		n/a	3	17	4	30	3	17	0	1
BackgroundPanel		94%		n/a	1	6	4	52	1	6	0	1
PromptingTextArea		100%		100%	0	22	0	53	0	15	0	1
PromptingTextArea.PromptFocusListener		100%		100%	0	5	0	7	0	3	0	1
View.ViewType		100%		n/a	0	1	0	4	0	1	0	1
SingleEntryView.State.new SingleEntryView.State() {...}		100%		n/a	0	2	0	6	0	2	0	1
SingleEntryView.State.new SingleEntryView.State() {...}		100%		n/a	0	2	0	6	0	2	0	1
SingleEntryView.State		100%		n/a	0	1	0	3	0	1	0	1
AbstractSwingView.new UndoableEditListener() {...}		100%		n/a	0	2	0	3	0	2	0	1
View.ViewType.new View.ViewType() {...}		100%		n/a	0	2	0	2	0	2	0	1
View.ViewType.new View.ViewType() {...}		100%		n/a	0	2	0	2	0	2	0	1
View.ViewType.new View.ViewType() {...}		100%		n/a	0	2	0	2	0	2	0	1
Total	947 of 3,752	74%	70 of 166	57%	110	268	254	891	61	184	2	26

Created with JaCoCo 0.8.6.202009150832

Figure 2: Code coverage for the `tab2xml.gui` package, where the GUI code is located. This report, and all statistics in the Front-End Tests section, are as of 2021-04-13.

Testing GUI code is a unique challenge, so it is unrealistic to get high code coverage of in GUI tests. However, TAB2XML's GUI has a coverage of 74 %, which is high for GUI tests. All of the code not tested in `tab2xml.gui` is very hard or impossible to test automatically. However, since almost all of these features are related to user interaction, errors can be found easily by manually running the application. In addition, most of the uncovered code is simple enough that it may not need testing. Much of the un-covered

code involves prompting the user with a dialog box, which cannot be tested automatically. Since each GUI functionality of TAB2XML that can be tested has a test, this testing is sufficient.

2.2 EditingPanelTest

This test tests the functionality of the `EditingPanel`, a custom `JPanel` that is used to edit XML metadata (by creating instances of the `XMLMetadata` class). These test cases were derived by making one test for each important functionality of the `EditingPanel`.

- `testCancel()`: Tests that the "Cancel" button works properly. It is implemented by inputting a text tab, setting a measure range, clicking Edit, editing the tab, then clicking Cancel. The test then ensures that the input text tab was not changed.
- `testComposer()`: Tests that the `EditingPanel` can correctly set the XMLMetadata's composer field. It is implemented by setting the text of the panel's composer entry, then getting the panel's resulting metadata, then checking the metadata's composer field. This is done twice: once for an empty field (the resulting metadata should have no composer) and once for a full field (the resulting metadata's composer should be the field's text).
- `testErrors()`: Tests various situations which could cause an error, such as having a negative measure start value. All error conditions are tested by creating the error conditions, simulating clicking a button, and using `assertThrows` or `assertDoesNotThrow` to test for an error (or no error). Many error situations are tested.
- `testMeasureRange1()`: Tests that empty text fields create the correct specified measure range for measure narrowing (first to last). It is implemented by clearing the `measureStart` and `measureEnd` fields and checking the resuling measure range.
- `testMeasureRange2()`: Tests that typing in a measure range is registered by the system. It is implemented by setting the text of the measure range fields and then checking the resulting measure range.
- `testNarrowing()`: Tests that the editing panel correctly calls the backend's measure narrowing functionality. Does not test that the narrowed text is correct, as that is the job of `MeasureNarrowingTest`. It is implemented by following a series of steps, using a `ViewBot` which is artificially connected to an `EditingPanel`:
  1. Load a text tab into the input text area, and edit the measure fields to select a measure range
  2. Ensure that the Edit button is enabled then simulate pressing it
  3. Ensure that the narrowed text tab is correctly loaded into the view's narrowing text box, and that the panel registers that editing is underway
  4. Use `view.setNarrowedText` to edit the narrowed text
  5. Ensure that the Done button is enabled then simulate pressing it
  6. Ensure that the view's input text is correctly changed, and that the panel registers that editing is complete
- `testTimeSignature()`: Tests that the EditingPanel correctly uses the time signature functionality. It is implemented by setting a measure range and time signature in the EditingPanel's text fields, then simulating pressing the Set Time Signature button. The panel's generated metadata is checked for the resulting time signature.

- `testTitle()`: Tests that the `EditingPanel` can correctly set the title field. It is implemented in the same way as `testComposer()`, except that the case with the empty field has a different output: the metadata should contain the default title "Untitled Score", instead of containing no title.

This testing is sufficient because every important functionality of the `EditingPanel` is tested. The code coverage of `EditingPanel` is 90 %. The `XMLMetadata` class, which is related to `EditingPanel`, has 100 % code coverage. The only code that is not covered by tests is code that should never run, and error code that doesn't finish running due to the way the tests are implemented (because the `ViewBot`'s `showErrorMessage` method is implemented by throwing an exception, the methods are marked as missed).

2.3 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. The Presenter has 72 % code coverage, and most of the missed lines are trivial error handling code.

2.4 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 getPromptFont(). 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. The code coverage for `PromptingTextArea` is 100 %.

2.5 TabbedViewUserExperinceTest

This test tests the user experience of the default `TabbedView`. These test cases were derived by considering the important functionality of the view. Use cases related to metadata

editing are covered by `EditingPanelTest`, so they are not tested here. All of the tests are implemented by using the view's package-private fields to simulate the user performing a specific use case, and then checking the correctness of the values or states of the view's fields.

- `testConvert()`: Tests the "Convert Text Tab" use case. The output musicXML is compared against output obtained from the backend, as testing the correctness of the output is the responsibility of the backend tests.
- `testInputButtonState()`: Tests the state of the "Convert", "Convert and Save" and "Save Input" buttons in multiple scenarios to ensure they are correctly enabled and disabled depending on the scenario.
- `testOutputButtonState()`: Tests the state of the "Save Output" button in multiple scenarios to ensure it is correctly enabled and disabled depending on the scenario.

This testing is sufficient because all important use cases are covered by either this test or `EditingPanelTest`. Code coverage of the `TabbedView` class is 79 %, and all of the missed instructions are trivial lambda instructions that do not need to be tested or things that are hard to test automatically but easy to test manually.

2.6 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 coverage for `XMLMetadata` (which has functionality other than the time signatures being tested) is 100 %. All instructions in `XMLMetadata` that relate to time signatures are covered by this test.

2.7 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.

- `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

All three of these tests are implemented by setting the parameter to some value, then comparing the value set to the value returned by the appropriate get method.

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.

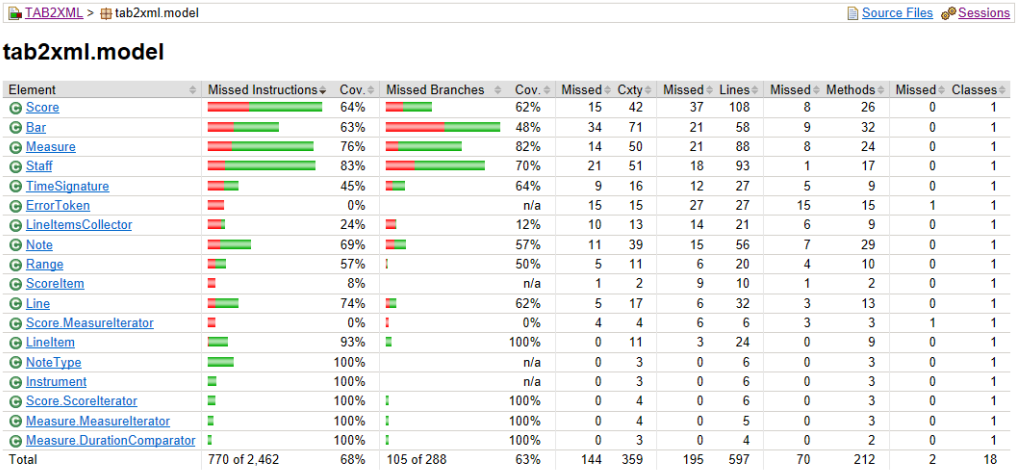
`View` has 18 % code coverage, `AbstractSwingView` has 47 % code coverage, and the three concrete view classes each have 80-90 % code coverage. `View`'s code coverage is low because many of its methods are default methods that are overriden by `AbstractSwingView`, plus one static factory method that does not need to be tested. `AbstractSwingView`'s code coverage is low because many of its methods cannot be tested automatically (though all can easily be tested manually).

3 Back-End Tests

3.1 Code Coverage and Test Sufficiency

Testing the Back-End is a very important part of ensuring that the system is working as intended. The coverage for the Back-End is solid overall except for areas which are either no longer used, have unimportant information, or have the testing covered by a different part of the system in their place. For example, many parts related to drums are either not used or were tested by a different part of the system since drums were implemented last, and are also simpler in comparison to guitar. The testing done for the Back-End was sufficient because many example combinations of different text tabs were tested and each example was different enough to cover all possibilities required to be converted by the system.

Code coverage reports for backend code (all as of 2021-04-13, as with all statistics in the Back-End Tests section):

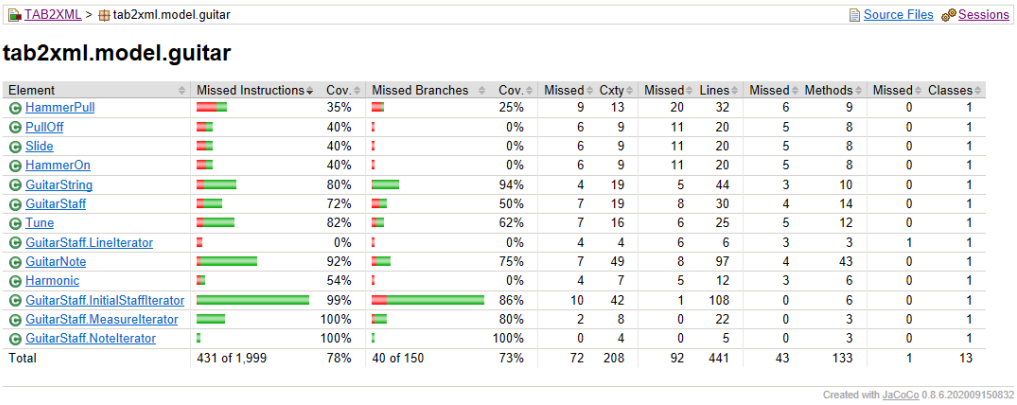


TAB2XML > tab2xml.model — Source Files — Sessions

tab2xml.model

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Score		64%		62%	15	42	37	108	8	26	0	1
Bar		63%		48%	34	71	21	58	9	32	0	1
Measure		76%		82%	14	50	21	88	8	24	0	1
Staff		83%		70%	21	51	18	93	1	17	0	1
TimeSignature		45%		64%	9	16	12	27	5	9	0	1
ErrorToken		0%		n/a	15	15	27	27	15	15	1	1
LineItemsCollector		24%		12%	10	13	14	21	6	9	0	1
Note		69%		57%	11	39	15	56	7	29	0	1
Range		57%		50%	5	11	6	20	4	10	0	1
ScoreItem		8%		n/a	1	2	9	10	1	2	0	1
Line		74%		62%	5	17	6	32	3	13	0	1
Score.MeasureIterator		0%		0%	4	4	6	6	3	3	1	1
LineItem		93%		100%	0	11	3	24	0	9	0	1
NoteType		100%		n/a	0	3	0	6	0	3	0	1
Instrument		100%		n/a	0	3	0	6	0	3	0	1
Score.ScoreIterator		100%		100%	0	4	0	6	0	3	0	1
Measure.MeasureIterator		100%		100%	0	4	0	5	0	3	0	1
Measure.DurationComparator		100%		100%	0	3	0	4	0	2	0	1
Total	770 of 2,462	68%	105 of 288	63%	144	359	195	597	70	212	2	18

Figure 3: Code coverage for `tab2xml.model`



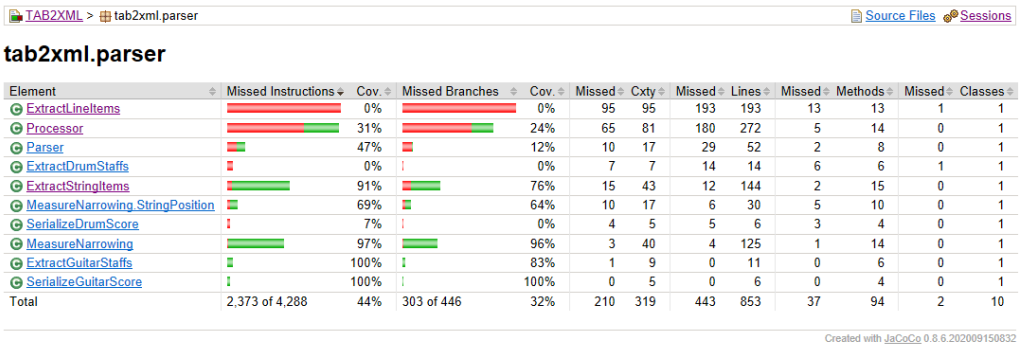
TAB2XML > tab2xml.model.guitar — Source Files — Sessions

tab2xml.model.guitar

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
HammerPull		35%		25%	9	13	20	32	6	9	0	1
PullOff		40%		0%	6	9	11	20	5	8	0	1
Slide		40%		0%	6	9	11	20	5	8	0	1
HammerOn		40%		0%	6	9	11	20	5	8	0	1
GuitarString		80%		94%	4	19	5	44	3	10	0	1
GuitarStaff		72%		50%	7	19	8	30	4	14	0	1
Tune		82%		62%	7	16	6	25	5	12	0	1
GuitarStaff.LineIterator		0%		0%	4	4	6	6	3	3	1	1
GuitarNote		92%		75%	7	49	8	97	4	43	0	1
Harmonic		54%		0%	4	7	5	12	3	6	0	1
GuitarStaff.InitialStaffIterator		99%		86%	10	42	1	108	0	6	0	1
GuitarStaff.MeasureIterator		100%		80%	2	8	0	22	0	3	0	1
GuitarStaff.NoteIterator		100%		100%	0	4	0	5	0	3	0	1
Total	431 of 1,999	78%	40 of 150	73%	72	208	92	441	43	133	1	13

Created with JaCoCo 0.8.6.202009150832

Figure 4: Code coverage for `tab2xml.model.guitar`



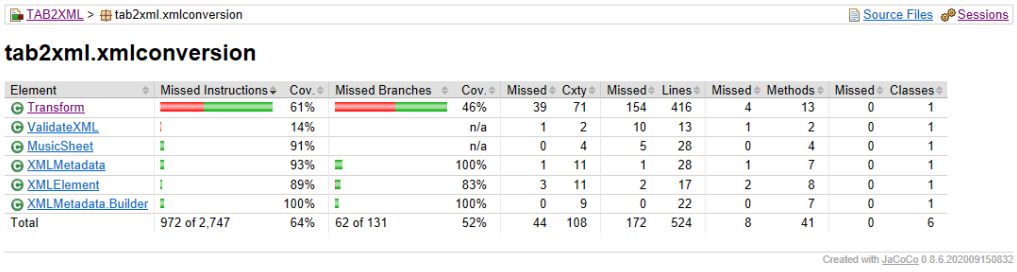
TAB2XML > tab2xml.parser — Source Files — Sessions

tab2xml.parser

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ExtractLineItems		0%		0%	95	95	193	193	13	13	1	1
Processor		31%		24%	65	81	180	272	5	14	0	1
Parser		47%		12%	10	17	29	52	2	8	0	1
ExtractDrumStaffs		0%		0%	7	7	14	14	6	6	1	1
ExtractStringItems		91%		76%	15	43	12	144	2	15	0	1
MeasureNarrowing.StringPosition		69%		64%	10	17	6	30	5	10	0	1
SerializeDrumScore		7%		0%	4	5	5	6	3	4	0	1
MeasureNarrowing		97%		96%	3	40	4	125	1	14	0	1
ExtractGuitarStaffs		100%		83%	1	9	0	11	0	6	0	1
SerializeGuitarScore		100%		100%	0	5	0	6	0	4	0	1
Total	2,373 of 4,288	44%	303 of 446	32%	210	319	443	853	37	94	2	10

Created with JaCoCo 0.8.6.202009150832

Figure 5: Code coverage for `tab2xml.parser`



TAB2XML > tab2xml.xmlconversion — Source Files — Sessions

tab2xml.xmlconversion

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Transform		61%		46%	39	71	154	416	4	13	0	1
ValidateXML		14%		n/a	1	2	10	13	1	2	0	1
MusicSheet		91%		n/a	0	4	5	28	0	4	0	1
XMLMetadata		93%		100%	1	11	1	28	1	7	0	1
XMLElement		89%		83%	3	11	2	17	2	8	0	1
XMLMetadata.Builder		100%		100%	0	9	0	22	0	7	0	1
Total	972 of 2,747	64%	62 of 131	52%	44	108	172	524	8	41	0	6

Created with JaCoCo 0.8.6.202009150832

Figure 6: Code coverage for `tab2xml.xmlConversion`

3.2 ParserTest

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

- `testGuitarConversion()`: Tests that Guitar conversion is handled as expected. This includes checking for the correct notes + amount of notes, correct number of strings, and correct number of measures. There is 5 versions of this test, as shown below, and each one includes different input files that handle different text tab cases. This test was created by taking an input of a guitar tab via text file, manually recording all of the expected information in an array and comparing that expected information with the information produced by the respective functions that are used in the xml conversion process.

- `testGuitarConversion_0()`: This test is a simple correctness test which tests the first tablature example provided during the project. This test makes sure that the correct notes are parsed with the correct attributes.

- `testGuitarConversion_1()`: This test makes sure that guitar tablature with multiple staffs are parsed correctly with the correct number of measures. This will ensure that more complex tablature will be correctly parsed and in the natural order of the notes.

- `testGuitarConversion_2()`: This test ensures that the notes within the different guitar actions such as hammer-on are parsed correctly. This test consists of the currently supported operations for guitar excluding grace notes and repeat sections (which will be tested in the next two guitar tests).

- `testGuitarConversion_3()`: This test ensures that grace notes function as expected. The grace note modifier acts greedy and collects any of the notes that follow the action. This test makes sure that this attribute works together with the notes natural ordering.

- `testGuitarConversion_4()`: This test ensures that a basic repeat section functions as expected.

- `testBassConversion()`: Tests that Bass conversion is handled as expected. This includes checking for the correct notes + amount of notes, correct number of strings, and correct number of measures. There is multiple versions of this test and each one includes different input files that handle different text tab cases, similar to the guitar cases above. This test was created by taking an input of a bass tab via text file, manually recording all of the expected information in an array and comparing that expected information with the information produced by the respective functions that are used in the xml conversion process.

- `testDrumsConversion()`: Tests that Drum conversion is handled as expected. This includes checking for the correct notes + amount of notes, correct octaves for each note, correct number of lines, and correct number of measures.

This test was created by taking an input of a drum tab via text file, manually recording all of the expected information in an array and comparing that expected information with the information produced by the respective functions that are used in the xml conversion process.

- `testGuitarScore()`: Tests that a guitar score object can be made successfully and have string and note objects assigned to it. This test was created by initializing a Score object, as well as GuitarString and GuitarNote objects, and adding the former two into the Score object. The purpose of this is to make sure that these objects are correctly created as they are the core of the xml conversion process.

- `testBassScore()`:Tests that a bass score object can be made successfully and have string and note objects assigned to it. This test was created by initializing a Score object, as well as GuitarString and GuitarNote objects, and adding the former two into the Score object. The purpose of this is to make sure that these objects are correctly created as they are the core of the xml conversion process. Tested with the common attributes of a bass score as opposed to the test above.

This test was created by initializing a Score object, as well as GuitarString and GuitarNote objects, and adding the former two into the Score object. The purpose of this is to make sure that these objects are correctly created as they are the core of the xml conversion process.

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. Different cases are handled by these tests to make sure that all the different types of notes are handled by the system.

3.3 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.

- `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 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.

** TestArea These tests were derived to make sure that the octaves of notes in a Guitar or Bass tab are converted correctly.

- `testGuitarOctaves_0()`: Tests that guitar octaves are correct.

This test was created by calling the getOctave() method for every fret on the guitar.

- `testBassOctaves_0()`: Tests that bass octaves are correct.

This test was created by calling the getOctave() method for every fret on the bass.

This is sufficient because every single fret on the instrument is tested.

3.4 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.
- `testLinearize2()`: Tests that linearization handles blank lines at the start and end correctly.
- `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 96 %. 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).