TAB2MXL

TEST PLAN

Version <2.0>
04/13/2021

VERSION HISTORY

Version #	Implemented By	Revision Date	Reason
1.0	Alborz Gharabaghi, Derui Liu, Isaiah Linares	02/28/2021	Test Plan draft
2.0	Alborz Gharabaghi, Derui Liu, Isaiah Linares, Lian Attily	04/13/2021	Finalized Testing Document

TABLE OF CONTENTS

1 INTRODUCTION

1.1 PURPOSE OF THE TEST PLAN DOCUMENT

This Testing Plan documents and tracks the necessary information required to effectively define the approach to be used in the testing of the project's product. Its intended audience for the customer of our project to demonstrate the many tests in our product. This document will provide JUnit 5 test cases to test the main components of the TAB2MXL software, which include GUI Testing, XML Class Testing, Drum Parsing Testing, Chain Class Testing and Guitar Tab Testing.

2 GUI TESTING

2.1 ITEMS TESTED

Item to Test	Test Description	Responsibility
TestConvertFile()	Tests if the software successfully produces an xml file in the location specified by the user	Lian
TestErrorMessages()	Tests robustness of software and that it will show dialogues that warn the user when the tab format is incorrect	Lian
TestUploadFile()	Tests the software's ability to upload a file and append it to the textarea	Lian
TestAdditionalAttributes()	Tests if adding additional attributes like title, composer, and lyricist read the correct input from the user	Lian
TestRequiredAttributes()	Tests if the required attributes like key signature, time signature, conversion type, and instrument type read the correct input from the user and warn the user if they do not select any. It also tests if the default (no selection) value is correct.	Lian
TestAdvancedSettings()	Tests if the advanced settings window pops up properly and tests if the text fields and combo boxes contain the correct information and function well.	Lian
TestSaveChanges()	Tests if the save changes prompts a dialogue	Lian
TestHelpMenuItems()	Tests if the menu item under "Help" contain the correct titles and functions properly (for instance, the "User Manual" menu item should open the user manual as a PDF file.	Lian
TestOpenRecentButton()	Tests if the Load Recent button in the menu bar under "File" loads the most recently saved tab and its attributes	Lian
TestExit()	Tests if exiting through the menu bar under "File" actually closes the platform	Lian
TestUploadButton()	Tests if the upload file button prompts a FileChooser	Lian

TestEmptySaveChangesButton()	Tests if clicking on the “save changes” button when the text area is empty prompts an error dialogue window.	Lian
ConversionTypeCheck()	Tests if the conversion type combo box contains both supported conversion format (Tab, Sheet Music)	Lian
InstumentTypeCheck()	Tests if the instruments combo box contains all three supported instruments (Drums, Guitar, and bass)	Lian

2.2 TEST APPROACH(S)

The GUI will be tested using TestFX, which is a GUI unit test framework for JavaFX applications. For more information on the TestFX API follow this link: https://github.com/TestFX/TestFX.

2.3 TESTING DERIVATION(S)

The unit tests were derived from the Main Controller class. The testing input is provided by manually pasting a sample guitar tab into the text area and performing numerous use cases on it.

2.4 TESTING SUFFICIENCY

These test cases prove to be sufficient as they cover all the code that must be covered, including pasting into the text area, editing text fields, picking from the combo box drop-down menus, as well as functionality of the different buttons and the menu bar items. Test coverage using Jacoco reports that the Main Controller has 60% coverage. The uncovered code falls into one two categories: *untaken catch statements and code that references file/uses a file chooser*. It is difficult to specify a valid tab input path on an arbitrary PC, and so these sections of code remain uncovered. The testing is sufficient as it covers almost all of the features of the GUI and asserts that they function as they should, and display the appropriate text to the user at all times.

3 XML CLASS TESTING

3.1 ITEMS TESTED

Test Name	Test Description	Responsibility
testChangeStep()	Changes step value in part object, checks to see if it actually changed.	Alborz
testNote()	Creates a note using the first constructor, checks to make sure every attribute is accurate to what was passed into the constructor.	Alborz
testNote2()	Creates a note using the second constructor (included notation object), makes sure every value added is correct. Changes values in the note (fret and string from notation) and checks to see if they changed properly.	Alborz

testPartConstructor()	Checks to make sure two part objects are not the same.	Alborz
testMeasure()	Creates a measure using an empty constructor, checks to make sure all attributes are empty. Sets a value to the measure, checks to make sure the set was made. Creates a second measure using a non-empty constructor, checks the attributes to make sure they are set to what they should be (based on object creation). Adds a note to the measure, changes duration of note and makes sure the change occurred successfully. Finally checks that both measure objects are different.	Alborz
testStaffDetails()	Tests new StaffDetails and StaffTuning objects. Creates a StaffTuning object using an empty constructor, adds a testing step and makes sure it was added properly. Creates new staff details object using empty constructor, adds StaffTuning object to it and makes sure the change happened properly.	Alborz
testInstrument()	Creates 3 new instrument objects and gives each one a name, checks to make sure the name actually shows up in the objects ID attribute.	Alborz
testSPW()	Tests the Score Partwise Writer (SPW), creates two new SPW objects that contain the exact same attributes. First checks to make sure both are not equal to each other, then checks to make sure attributes were properly set from the constructor. Sets a new Part object to the Score Partwise object, then checks to make sure another Part object and the Score Partwise Part object are not the same.	Alborz
testPW()	Tests the Part Writer (PW) class to ensure a proper part is being written. It makes two part writer objects and sets one object's partID to "test". It then checks to make sure this happened. It then compares the two part object's partID's are different.	Alborz
testDPW()	Tests the Drum Part Writer (DPW) class to ensure a proper drum part is being written. It makes two drum part writer objects and sets one object's partID to "test". It then checks to make sure this happened. It then also to see if adding backups and forwards work, as well as the next measure method. It also creates a drum note and sees if the attributes added to the drum note are equal.	Alborz

3.2 TESTING APPROACH

The XML classes will be tested using unit tests with JUnit 5 framework. These include various tests like assertEquals, assertNotEquals, assertNull, assertNotNull etc. Our tests will be using pre made objects from various constructors and then include object manipulation to ensure they act as they should.

3.3 TESTING DERIVATION(S)

These various tests were derived from the countless XML classes present in our project. With over 36 XML classes, we wanted to make sure they are acting as they should be when created and fed values. If these classes are not returning and storing the right values, our XML output will not be correct. The test cases include using the different constructors for each class, using get and set methods and making sure those methods are actually working. We also have "writer" classes that take in information from the "Chain" class and write the correct XML objects; these need extensive testing. If these "writer" classes are not actually creating the right objects, our program will face many bugs in output. We tested about 1/3rd of the regular XML class objects as they all follow the exact same format, with getters and setters being mainly tested. The writer tests included creating a new writer object, and emulating how the Chain class will be using this object to ensure it outputs the correct information.

3.4 TESTING SUFFICIENCY

The XML classes are mostly the same, with get and set methods and a few constructors. This means that we don't need to be too exhaustive with these tests as every class is not very complex, nor different from one another. The only way our test cases would result in a failure would be someone directly changing the class itself, which is highly unlikely. Our test cases cover about 1/3rd of the XML classes so far which should be more than sufficient for the reasons stated above. In terms of the "writer" class test cases, they are sufficient because they cover the main cases in which they will be used. The Chain class uses these writer classes to make all of the XML objects, our tests emulate this process to ensure it is running correctly at all times.

4 DRUM TAB PARSING TESTING

4.1 ITEMS TESTED

Test Name	Test Description	Responsibility
tabReaderDumkitTest()	This test ensures the correct functionality of the initializeDrumkit() method as well as the getDrumKit method in the DrumReader class. The professor's provided example is loaded as input, the test then checks that when a DrumReader instance is instantiated with the input measure that the initializeDrumkit records each instrument ID correctly. Using the gerDumKit method, the recorded kit is compared with the expected kit.	Isaiah
profExampleTest()	This test ensures the DrumReader class correctly parses the first measure of the prof's given example shown below. The methods setMeasure(), readNotes(), initializeDrumKit(), readColumn, adn	Isaiah

	hasNext must all work together correctly in order for this test to pass. In order to do this the expected values of each note in the first measure are compared to the parsed values, in the order that the column method parsing is supposed to have. Some of the important functionality which is tested is: beams, beams across different instruments, drums as a separate voice (excluded from other beams as well as distinct stem), notes on the same column, correct note data (step, octave, duration, voice type, notepad, beam and stem), and maximum beam length.	
openHiHatTest ()	This test is similar to the profExampleTest() except that it specifically tests that any High-Hat note with an ‘o’ notehead must have its ID changed to “P1-I47” which represents an open Hi-Hat.	Isaiah
rowParsingTest()	This tests the functionality of the not currently used, row by row parsing methods in the drum reader class, readNoteRow(), readRow(), and hasNextRow(). In order to test this the professor’s given example is loaded and used as input to generate all the note data for both measures. The note data is then outputted to the console, which is also recorded on the test document in the build/reports/tests/test directory.	Isaiah

4.2 TESTING APPROACH

The DrumReader will be Tested using the JUNIT 5 framework using assertions as well using deliverables to test information parsing.

4.3 TEST DERIVATIONS

The tests described above derived from the different forms of drum input that the DrumReader class is required to handle. They were made to cover any input that would be expected to be passed to the DrumReader class to be parsed. This also helps in the development of new features, automatically ensuring that all implemented features are still correctly functioning with different input.

4.4 TESTING SUFFICIENCY

The drum tests we have cover 96.4% of the DrumReader Class with 50 missed instructions, which covers branches that most drum tab inputs will need. Only a subset of inputs which were able to be read and passed to the drum parsing class will be accepted as valid input.

5 GUITAR TAB PARSING TESTING

5.1 ITEMS TESTED

Test Name	Test Description	Responsibility
basicTabTestCase()	The readNotes() method is integral to the parsing of guitar, bass, and drum notes. It returns information for a column of notes in a guitar measure. In order to test this a test file "testTab.txt" in the src/main/java folder is separated into measures using the TabReaderV4 class and then parsed with the Measure Reader class used for parsing guitar and bass tabs. The info parsed from each note is then stored in a string to be output so we can assure the info was collected correctly.	Derui

5.2 TESTING APPROACH

The MeasureReader class which parses guitar tabs is tested with the JUNIT 5 framework. We use the assertions to assure there are no errors when using the tab reader or the measure reader.

5.3 TEST DERIVATIONS

This test was derived from the many different features of a guitar tab, information on the notes, measures and techniques is normally output in the console to check for problems. This output contains the: note length, step, octave, alter, accidental, string and fret and much more information. Corroborating this information with the tab is vital to assure the correctness of the parsing functionality. It is also critical to ensure that the parser does not encounter issues parsing the various features that could possibly be present in a tab.

5.4 TESTING SUFFICIENCY

The test is able to confirm an entire song that contains all of the supported features is parsed correctly and that the parser does not encounter any issues parsing the various features.

6 CHAIN CLASS TESTING

6.1 ITEMS TESTED

Test Name	Test Description	Responsibility
testChainConstructor()	This test checks that a new chain object has the correct attributes that were fed into it.	Robert
testChainT2P()	This test checks the TABtoPART method in the chain class.	Alborz

	This method checks what instrument is selected and sets the staff line correspondingly. In the test I call this method, then check to make sure that the stafflines was set to 6, since the instrument selected was guitar. It then does the same for a bass instrument, and checks that stafflines is 4.	
testLyricist()	This test checks to see if the lyricist was properly instantiated in the chain constructor	Alborz
testComposer()	This test checks to see if the composer was properly instantiated in the chain constructor	Robert
testStaffLines()	This test checks to see if the staff lines were properly instantiated in the chain constructor.	Alborz
testChaindrumT2P()	This test checks to see if the drum reader can be properly built.	Robert
testChainbassT2P()	This test checks to see if bass can properly be created and tuning information can be fetched.	Robert

6.2 TESTING APPROACH

The chain class will be tested using JUnit 5 and will be tested to make sure it is chaining together XML objects properly through getter and setter methods.

6.3 TEST DERIVATIONS

These tests were derived based on the fact that Chain creates and sets many variables that are all very important in creating a proper XML output. Testing that the constructor works is clearly important as there are many variables in Chain, so ensuring they are set properly is key. Also making sure that the tabtopart() method works is important as it sets the base for tab parsing. If it does not work accordingly, tab parsing will not work.

6.4 TESTING SUFFICIENCY

These tests are sufficient at the moment because they test the biggest factors in *Chain*. It ensures that all variables are set properly, and that calling the tabtopart method works properly. Although Chain has other methods that start the conversion of a tab, the Drum Parsing Tests and Guitar Parsing Tests cover these cases fully, so there would be no urgent need to include these at the moment.