TAB2MXL

TEST PLAN

Version <1.0>
02/28/2021

VERSION HISTORY

Version #	Implemented By	Revision Date	Reason
1.0	Alborz Gharabaghi, Derui Liu, Isaiah Linares	02/28/2021	Test Plan draft

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 (TO BE IMPLEMENTED)

2.1 ITEMS TO BE TESTED / NOT 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

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.

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

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

Item to Test	Test Description	Responsibility
initializeDrumKit()	This method must be tested to assure that the DrumReader class will work with different sets of drum kits. In order to test this the example provided by the prof is used to initialize a DrumReader Object. We then use the getter method to obtain the ArrayList of drum kit parts created and compare them with the correct results hardcoded in the test. To do this we use the method assertLinesMatch to ensure the drumReader is instantiated with the correct drumKit.	Isaiah
readNote()	This method is integral to the drum reader so that it solves the problem of unparsed tabs. This Method will parse whole measures and extract essential information about each note found. This method must be tested to	Isaiah

	ensure that the correct note information is returned.	
readColumn ()	This method helps isolate one column of notes from the measure to be parsed. And needs to be tested to assure the notes collected match the measure.	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

In the airInTonightTest where theand the measureOutputTest all information about the notes in each measure is stored in a string and outputted to the console so we are able to check if the right information returned.

4.4 TESTING SUFFICIENCY

All aspects of the DrumReader are tested to ensure the correct format is output for certain test cases. Although the DrumReader is designed to have leniency for differently formatted input which means it is not possible to test all cases since Drum Tabs come in many formats. This class also works in conjunction with the tab reader which may not handle all inputs possible correctly.

5 GUITAR TAB PARSING TESTING

5.1 ITEMS TESTED

Item to Test	Test Description	Responsibility
getNotes()	This method is integral to the parsing of guitar 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 MeasureReaderV3 class. 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

The output string file is used to store all information of each note in the song which we output in the console to check for problems. This output contains strings of the: notelength, step, octave, alter, accidental, and fret of each note in the tested tab which assures that the getNotes() method has the correct functionality.

5.4 TESTING SUFFICIENCY

The test is able to confirm an entire song is parsed correctly and that the information for each note is the same as the tab.

6 CHAIN CLASS TESTING

6.1 ITEMS TESTED

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

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 because 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 now 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.