TabToPDF
Test Document

Group 1

Tom, Khurram, Saad, Ron, Skyler, Alex, Jon, Bilal

CSE2311 Prof. Tzerpos

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Purpose

The purpose of this document is to describe the testing performed on TabToPDF.

1.2 Scope

This document will cover test cases and JUnit testing of TabToPDF. Figures of each will be displayed and the derivation and implementation will be covered. Testing coverage of the software and what was not tested will be discussed.

1.3 Overview

An overview of all the testing will be shown. Test cases and then actual outputs followed by the JUnit test suites are visualized. The sufficiency and coverage of the testing follows.

1.4 Reference Material

Please refer to the Design, Requirements and User Manual documents for more details.

1.5 Definitions and Acronyms

TabToPDF – stands for tablature to PDF format and is the name of the software
GUI – Graphical User Interface
MVC – Model View Controller design
Conversion – conversion code component for text file to PDF format
Component – a collection of classes that contribute to a certain function in the software

2. TEST OVERVIEW

Testing consists of two parts. One part is the test cases, of which there are 11. The other part is the test suite which consists of more than 100 methods among all the classes tested organized into converter and formatter test suites.

An entire built in error system exists in the software called the formatter which tests input text files and corrects them before they are converted detailed in the Design document.

3. TEST ARCHITECTURE AND DESIGN

In this section, all the test cases are shown with the test suites. How the formatter and convertor which formats and converts the user input were tested is explained. We will go over how we derived the ideas and how we actually implemented them. We will also talk about why our testing is sufficient and also discuss our testing coverage.

3.1 Test Cases, JUnit Test Classes

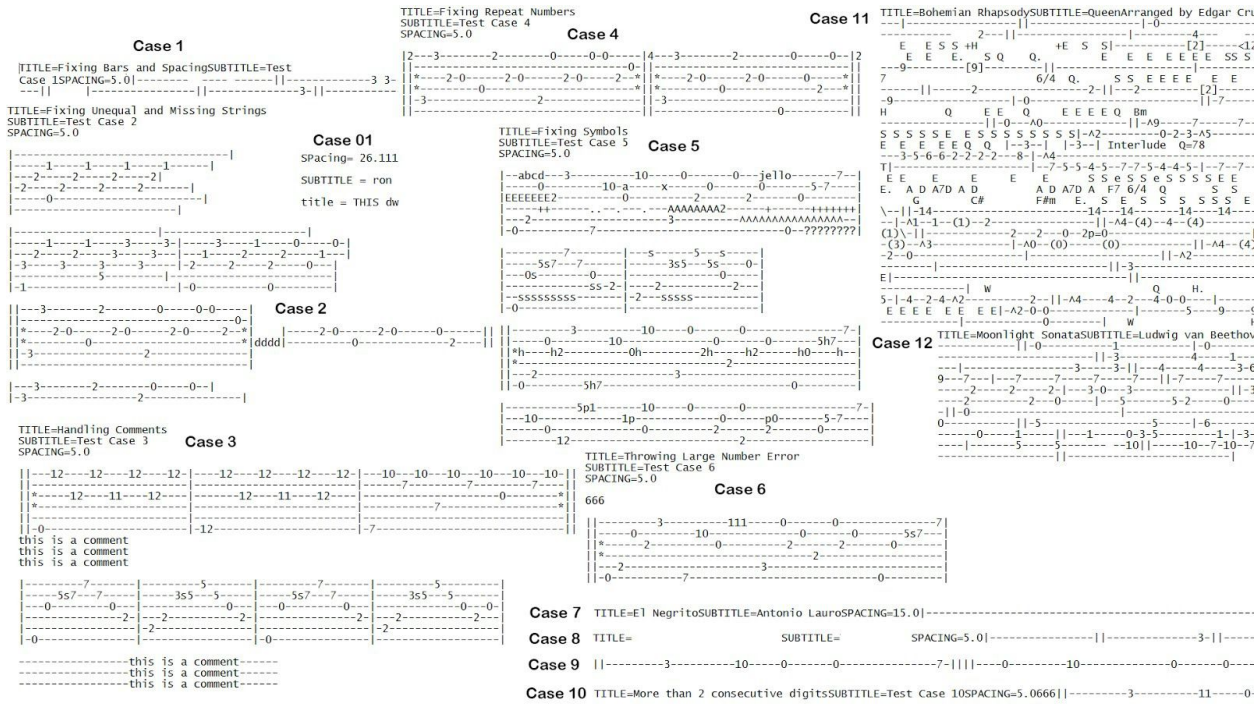

Figure 1: Test cases in text format. 0-11 cases exist. 0 is not shown because it is an empty file.

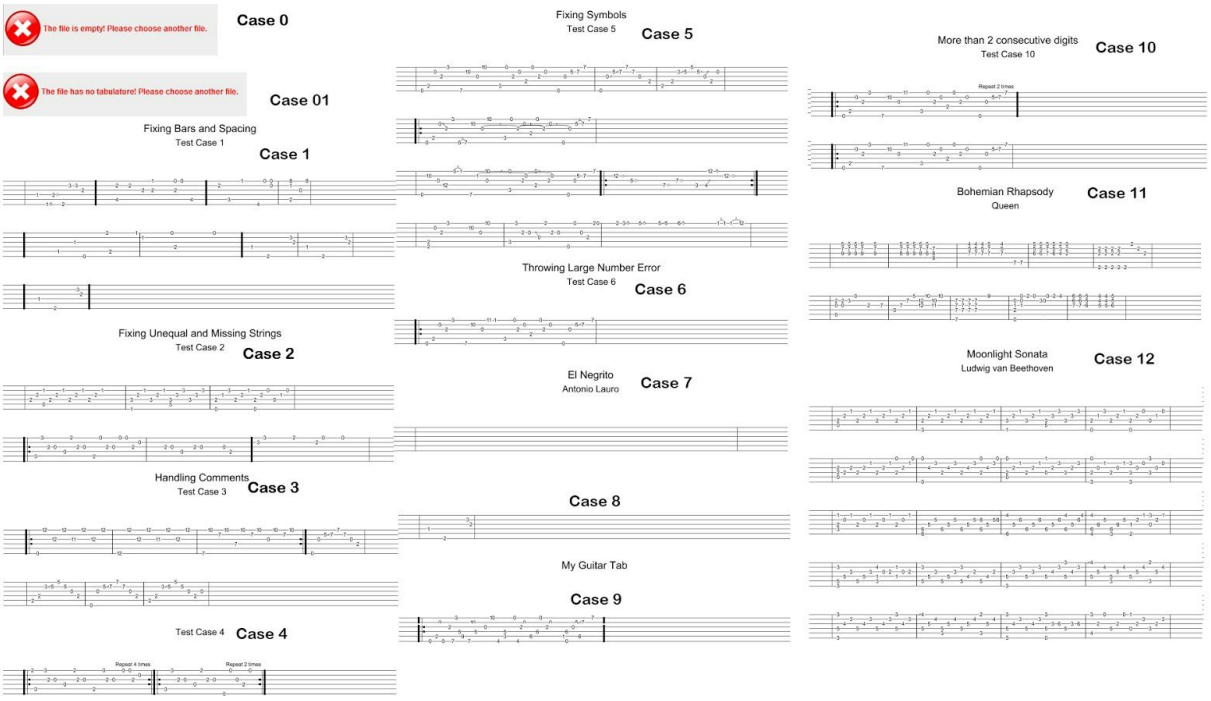

Figure 2: Test case actual outputs from software. Case 0 is included.

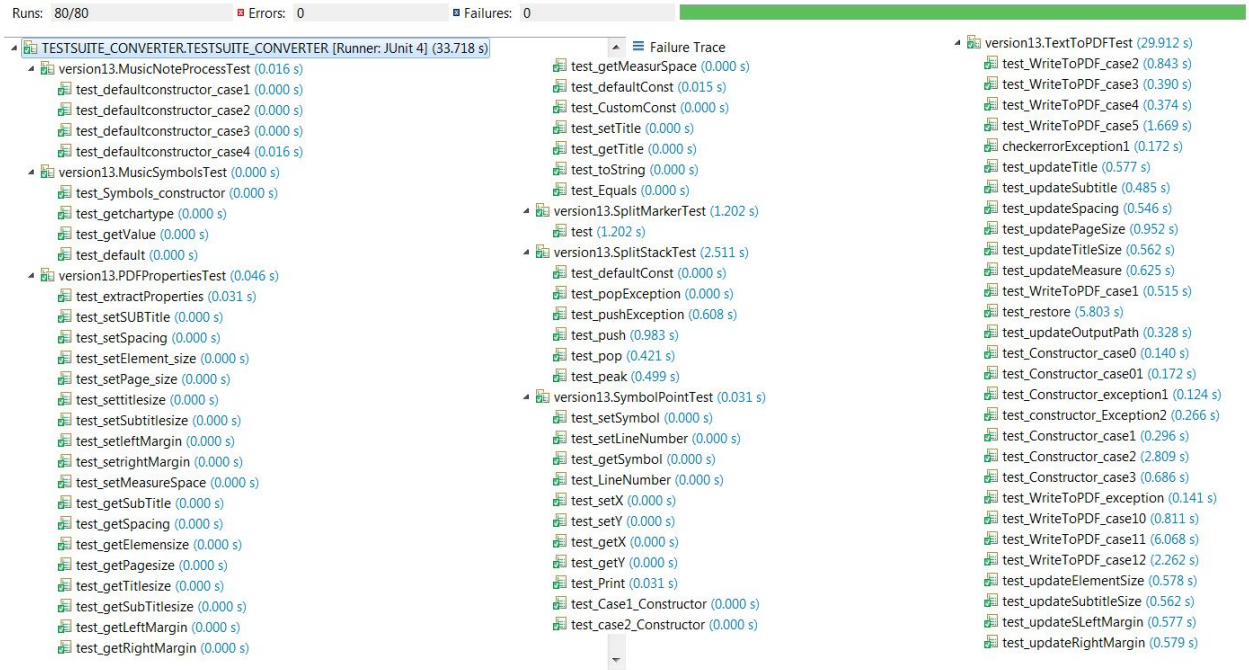

Figure 3: Converter Test Suite.

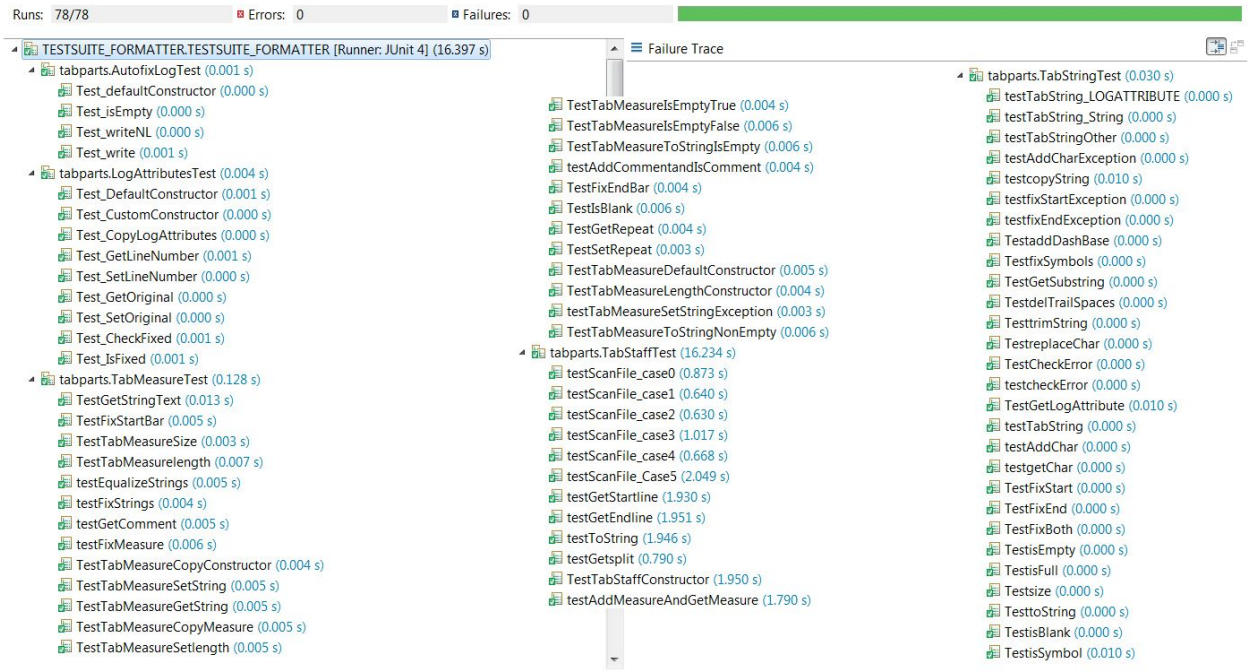

Figure 4: Formatter Test Suite.

3.2 Derivation and Implementation

The purpose of testing the formatter was to check if it can handle all the possible cases. To design these cases, our testing department of the team got together and made bunch of cases with unformatted input. We exhausted all the cases and came with 5 cases which cover all the cases and put them into 5 separate files. Each of the file had a special case and a new challenge for our formatter. In these cases we had everything from gibberish to English.

Along with the six unformatted files, we created six more files which would show how these unformatted files are supposed to look after they have been formatted.

Below is the contents of our six unformatted files along with their formatted versions.

NOTE: The formatted version differentiates tab string by comma ',' and tab measure by square bracket '[]'.

File 1: This File was meant to test if our formatter can fix missing bars and spaces which the user may have mistyped.

```
|-------- ---- ------|
|-------------3 3-----|
|-------- ---- ---2 --|
|--<1 > <2>---- ------|
|------------ ------|
|-----<1> < 2>- ----
```

Formatted Version:

```
[[|---------------------|,
|-------------3-3-----|,
|-----------------2---|,
|---1---<2>-----------|,
|---------------------|,
|-----<1>---2---------|],
```

File 2: This File was meant to test if our formatter can fix unequal and missing strings.

```
|---------------------------------|
|-----1-----1-----1-----1------|
|---2-----2-----2-----2|
|-2-----2-----2-----2-------|
|-----0----------------------|
|-------------------------|
|-----2-0------2-0------0------||
|----------0--------------2----||
```

Formatted Version:

```
[|-----------------------------|,
|-----1-----1-----1-----1------|,
|---2-----2-----2-----2--------|,
|-2-----2-----2-----2----------|,
|-----0----------------------|,
|-----------------------------|]
[|-----------------------------||,
|-----------------------------||,
|-----2-0------2-0------0------||,
|----------0--------------2----||,
|-----------------------------||,
|-----------------------------||]
```

File 3: This File was meant to test if our formatter can recognize and handle comments.

```
||---12----12----12----12-|----12----12----12----12-|---10---10---10---10---10---10-||
||------------------------|-------------------------|------7---------7---------7----||
||*-----12----11----12----|-------12----11----12----|--------------------0--------*||
||*------------------------|-------------------------|-----------7------------------*||
||-------------------------|-------------------------|------------------------------||
||-0----------------------|-12----------------------|-7----------------------------||
this is a comment
this is a comment
this is a comment
```

Formatted Version:

```
[||---12----12----12----12-|,
||------------------------|,
||*-----12----11----12----|,
||*------------------------|,
||-------------------------|,
||-0----------------------|],
[|----12----12----12----12-|,
|-------------------------|,
|-------12----11----12----|,
|-------------------------|,
|-------------------------|,
|-12----------------------|],
[|---10---10---10---10---10---10-||,
|------7---------7---------7----||,
|---------------------0--------*||,
|-----------7------------------*||,
|--------------------------------||,
|-7------------------------------||]
```

File 4: This File was meant to test if our formatter can recognize and fix repeat numbers.

```
|2---3--------2--------0-----0-0-----|4---3--------2--------0-----0--|2
||-----------------------------------0-||-------------------------------||
||*----2-0------2-0------2-0-----2--*||*----2-0------2-0------0-----*||
||*---------0----------------------*||*---------0--------------2---*||
||-3-----------------2---------------||-3----------------------------||
||-----------------------------------||-------------------0----------||
```

Formatted Version:

```
[||2---3--------2--------0-----0-0-----||,
||-------------------------------0--||,
||*----2-0------2-0------2-0-----2---*||,
||*---------0-----------------------*||,
||-3-----------------2----------------||,
||-------------------------------------||],
[||---3--------2--------0-----0--||,
||-----------------------------||,
||*----2-0------2-0------0-----*||,
 ||*---------0--------------2---*||,
||-3---------------------------||,
||------------------0----------||]]
```

File 5: This File was meant to recognize all invalid symbols and delete them. It was also meant to make sure that the correct symbols are in the correct syntax, for example a harmonic is around <>, hammer, slide, pull are attached to a number e.g h2.

TITLE=Fixing Symbols

SUBTITLE=Test Case 5

SPACING=5.0

```
|--abcd---3----------10-----0-------0---jello-------7--|
|-----0---------10-a-----x------0-------0-------5-7----|
|EEEEEEE2----------0----------2-------2-------0--------|
|-----++------..--.---.---AAAAAAAA2------+------+++++++|
|---2---------------------3----------^^^^^^^^^^^^^^^^--|
|-0-----------7----------------------------0--????????|

|---------7--------|
|-----5s7---7------|
|---0s--------0----|
|-------------ss-2-|
|--sssssssss-------|
|-0----------------|
```

Formatted Version:

```
[|---------3----------10-----0-------0---------------7--|,
|-----0---------10--------------0-------0-------5-7----|,
|-------2----------0----------2-------2-------0--------|,
|---------------------------------2--------------------|,
|---2---------------------3----------------------------|,
|-0-----------7-----------------------------0----------|],
[|---------7--------|,
|-----5s7---7------|,
|---0s--------0----|,
|----------------2-|,
|------------------|,
|-0----------------|]
```

Now the next part was to implement the above idea in java and to also test it using JUnit. Now that we have both files, unformatted file and the formatted version of it. We wanted to compare them and see if the formatted file is equal to what our formatter returns after it formats the unformatted file.

Before we start, in the formatter, there is a method called scanFile(File f) which plays the role of the formatter. It takes in a file as a parameter and formats it.

First We copy the content of our formatted version into a string called 'h' which we typed before calling the scanfile method and then we call the method scanFile with unformatted file as its parameter. We store everything in the staff. Then we store what the scanFile returns in a string called 'j'. At the end we compare our expected string (h) with our resulted string which the scanFile method returns (j) using assertTrue method and equals method for strings.

Below is the code for one of the cases.

```
@Test
        public void testScanFile_case4() throws Exception {
                File f = new File("inputfiles/case4.txt");
                tabs1.scanFile(f);
                String h = "[[||2---3--------2--------0-----0-0-----||, ||-------------------------------0--||,
" +  "||*----2-0------2-0------2-0-----2---*||, ||*---------0----------------------*||, " +
"||-3----------------2---------------||, ||-----------------------------------||], " +
"[||---3--------2--------0-----0--||, ||-----------------------------||," +
" ||*----2-0------2-0------0-----*||, ||*---------0-------------2---*||, " +
"||-3--------------------------||, ||------------------0----------||]]"  ;
```

```
        String j = tabs1.getList().toString();
        assertTrue(h.equals(j));
    }
```

MusicSymbolsTest.java (Test for MusicSymbols.java)

The cases that were taken were not random because the constructor only deals with defined enums.

Enums are:

```
enum Symbols {
    OneBar_begin,OneBar_end,
    OneBar_begin_lastline,OneBar_end_lastline,
    Two_Bar_begin,Two_Bar_end,
    Two_Bar_begin_lastline,Two_Bar_end_lastline,
    Three_Bar_End,Dash,
    One_digit,Two_digit,
    Slide,Pull,Hammer,
    Left_Half_Diamond,Right_half_Diamond,
    Star_Begin, Star_End,
    End_music_line,End_music_note,Space
}
```

Constructors:

This class has only two constructors one takes an enum symbol and int value and other just takes the enum symbol, the way we tested this is by

Constructor is initialized with arguments. A value is assigned and assertequals is called to test whether the expected value matches the actual output number. the attributes if they are assigned properly and correctly or not, same thing with the constructor 2 except the value is automatically set to 1 so just a known value expected is tested which is 1 and the symbol tested.

Accessors:

Expected value was chosen and the accessor method returned a value to compare. For example, there were two accessor methods getCharType() which returned the music symbol and getValue() which returned an int.

Conclusion:

All the possible code was covered in this class. Since this class was small the test cases used were sufficient. Testing coverage for this class was 98.1% although its shows all the line of code is covered.

MusicNoteProcessTest (Test for MusicNoteProcess.java)

Since this class contains the list of strings that are going to be processed into symbols, the way this class works is that it gets in a string of valid format. The type of symbol is stored and counts the quantity so that when drawing, the draw class knows how many symbols to draw and which symbols to draw.

The cases were all valid tabStrings (already fixed) and passed in and checked whether the amount of a specific symbol was stored correctly or not.

For example ||*h-----2s-----2p-----<2>-----2-------||: will recognize the type of bar in the start and count it and use an enum from MusicSymbols.java class. Then the asterisk would be stored and counted and so on...

The cases used were with double bars, single bars ,start and end asterisk, etc., until all of the cases would be covered by the variables in the enums.

Constructors:

Since the constructor assigns a new list to the attribute and then calls a method called stringProcess(), test cases for this constructor were made to check if the strings were being processed properly into music symbols. This way this was tested was that a toString() method was made for this class specifically to help the testers that would return a whole list of what symbol was used and the number of that symbols. An expected String was compared it with the toString() output of the class.

For example: |--2----3----| the toString method would return

[OneBar_begin_lastline,1]+[Dash,2]+[One_digit,2]+[Dash,4]+[One_digit,3]+[Dash,4]+[OneBar_end_lastline,1]+[End_music_line,1]+[End_music_note,0]+

The first bar is counted and the enum name is returned along with the value separated by comma

surrounded by the solid square brackets and finally the dases are read .

Then it is repeated, [enum,value]+[another enum, value]+...

Most of the possible cases were defined by the limited amounts of enums that this class recognizes which build upon the MusicSymbols.java class.

Testing Coverage for this class was 100% since nothing was missed.

PDFPropertiesTest (test for PDFProperties.java)

This class is supposed to provide the TextToPDF.java class and DrawClass with properties of the PDF document that is to be created. This class gets some properties from the text file and the rest is assigned by default or generated. Properties which are extracted from the text file is Title, Subtitle and spacing, and the rest is assigned by this class such as PAGESIZE, Symbol Size and so on.

Constructors:

There were two constructors, a default constructor and an overloaded constructor that takes in a String title, a String subtitle and a float for spacing.

The way it was constructed is by looking at what values the default constructor assigns to the attributes of the object and then the default constructor was called in the test case. Expected values were compared with actual.

The overloaded constructor was tested by the same way the default constructor was tested except the title, subtitle and spacing were passed and compared. However, since the assertEquals method is deprecated for float types, int type had to be cast.

Methods:

There was one method that was supposed to extract the title, subtitle and spacing from text file. This was tested by going into the file and noting down the title subtitle and spacing and then calling the extractProperties method. This method takes in a file and then checks the attributes title, subtitle and spacing of the object if they were the same as in the text file and after the method call.

Another method was the toString method. This was tested by typing out the expected string which will be outputted and then using assertTrue and using equals method of strings.

Equals method of this class was tested by whether the two objects were equal then the attributes would be compared and they would be identical.

The rest of the methods were accessors and mutators which retrieve or change the values of the

attributes. They were tested by calling the accessor and comparing with the expected return and mutators were tested similarly calling the mutator and then checking if the attribute value was set by that mutator and then was compared with the value assigned by the mutator.

Different values were passed into constructor and methods for sufficient testing.

The coverage for this class was 90.9% because some exceptions were missed.

SymbolPointTest (test for SymbolPoint.java)

This class would contain the info about the symbol such as coordinates, the symbol itself and the line number which are attributes that exist in the object SymbolPoint.

Different values of attributes were passed to the object. The accessors and mutators were called to see if the test cases test the class properly.

Since this was a small class not much effort was needed to ensure all the cases had gone through.

Constructors:

There was only one constructor and that constructor takes in four arguments that are the symbol. which is passed from the string. float x and y coordinates of the symbol and int line number. The constructor uses the mutator methods to assign values to the object's attributes.

A new object is made using this constructor and arguments are passed into this constructor and assertEquals was used to compare expected with actual.

Methods:

There was a print method in this class which prints a string to the console containing information about the Symbol Point object (the values of the attributes of the object). The way we tested this is that we change the output Stream to a file so that this will write the message to a file and we then get the content of the file as a string and then comparing that string with the expected string which we made and then using assertTrue and equals method of Strings.

The rest of the methods were accessors and mutators which retrieve or change the values of the attributes and they were tested by calling the accessor and comparing with the expected return and mutators were tested similarly calling the mutator and then checking if the attribute value was set by that mutator and then was compared with the value assigned by the mutator.

The cases used were sufficient for such a small class different values were tried different symbols and kept using the same object also so that it goes through many methods so that it can cover previous instructions.

The coverage for this class was 100% no instructions missed.

TextToPDF (test for TextToPDF.java)

This class contributes a lot to the conversion code. This class is responsible for creating an object TextToPDF which contains properties of the PDF from PDFProperties.java and input path which is the path of the text file to be converted. An output path where the pdf file will be stored. Tabstaff object is used to get rid of errors and format the input file (by using scanfile method in tabstaff class).

Constructors:

There is only one constructor in this class, which just takes in two arguments an input path and output path strings. Then inside the constructor it calls PDFProperties class with a method called extractProperties and assigns the value to its attribute and uses scanfile from tabstaff. This was tested by creating a PDFProperties object which will have the expected attributes such as title, subtitle etc and compare it with the attribute properties from the TextToPDF class and check if they are equal using equals method in PDFProperties class. Also input paths and output paths were tested to check if they are assigned to their respective attributes of TextToPDF object.

Methods:

This class has a method called WriteToPDF which is where the actual conversion is happening. This method also writes a log file with corrections and also displays a message that the file has been converted successfully. It was checked whether the file is converted. If the log file is not empty and the message will is displayed, then the conversion is successfully completed and the PDF is checked. The print statement was tested by changing the output stream to a file so that this will write the message to a file and then the content of the file as a string and then comparing that string with the expected string which was made using assertTrue and equals method of Strings.

A method called checkInputErrors exists that just throws error exceptions. These exceptions were handled and tested.

the rest of the methods were accessors and mutators which retrieve or change the values of the attributes and they were tested by calling the accessor and comparing with the expected return and mutators were tested similarly calling the mutator and then checking if the attribute value was set by that mutator and then was compared with the value assigned by the mutator.

A lot of cases were done such as passing in a file that doesn't exist and an empty file, a file that doesn't have music content or if an invalid input path has been passed.

The classes such as DrawClass were automatically tested when the TextToPDF class was

tested and when the MusicSymbols and MusicProcess were tested.

4. SUFFICIENCY AND FEATURES NOT TESTED

The testing done for the formatter is sufficient because the five cases contain all the possible cases. This testing makes sure that the formatter is doing its job correctly and efficiently. This testing also helped in finding some cases which the older version of the formatter could not handle so a newer version of the formatter had to be made and it was checked again and it worked. Therefore, since this testing covers all the cases and more bugs were found in our formatter which were fixed, the testing proved to be sufficient.

The testing was sufficient for conversion because almost all of the code had been reached and behaved like it was supposed to.

Only some of the exceptions were missed due to if and else statements and unreachable code.

Overall, the tests performed are sufficient for many reasons. These reasons include the fact that almost all cases are covered and almost all classes are tested. The worst inputs were created and put through the software and an output PDF would still be displayed. Should the user create input that is truly unformattable or strays too far from input requirements, the software will notify the user of the formatting errors.

The reasons why all cases and classes are not tested is due to the consideration that 100% testing may not be possible. Some lines of code are not testable and obscure test cases would likely be missed by a human. Some of the things that were not tested fully are the GUI, but even then rudimentary testing of all possible button presses was performed.

5. TEST COVERAGE

The Converter Package CODE COVERAGE IS 87.5%

The Formatter Package CODE COVERAGE IS 88.4%

Total COVERAGE WITHOUT GUI 96.3%

Total COVERAGE WITH GUI 73.8%

The GUI package coverage 40.7%

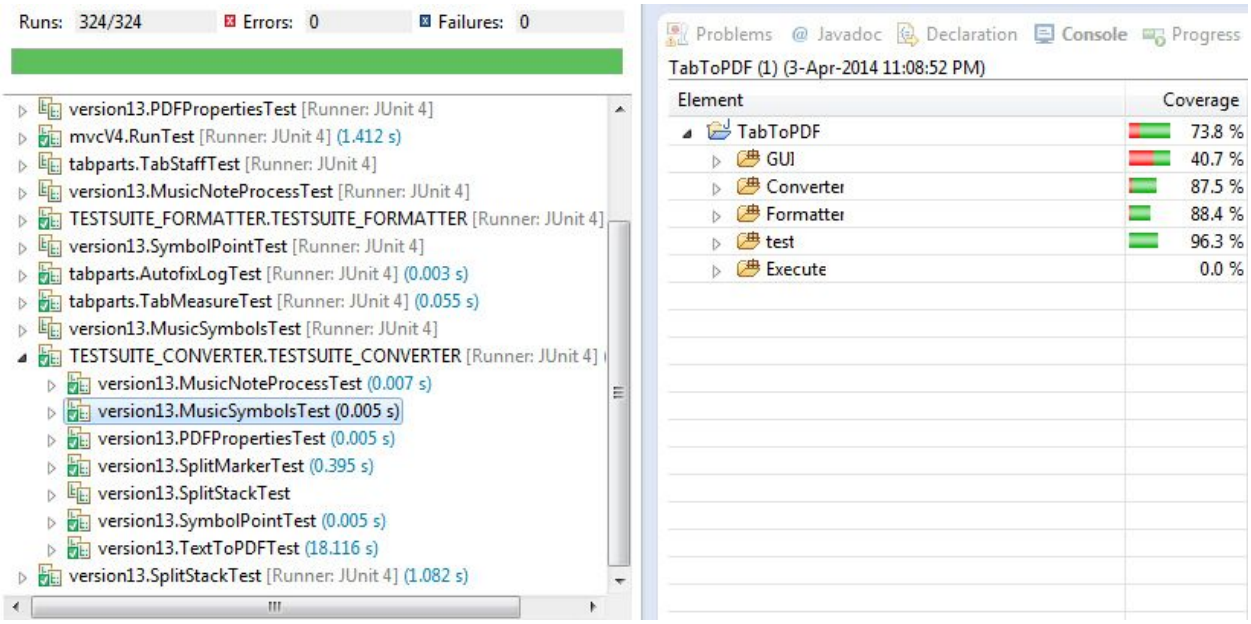

Figure 5: Final testing metrics.