Cse 2311 Software Development Project

Tab2PDF

Testing Documentation

York University

Professor Vassillios Bil Tzerpos

Team 3

Mohammad Nasar

Sirag Rauff

Waleed Azhar

Jason Kuffour

Calvin Tran

Umer Zahoor

Muhammad Shah

Table of Contents

1. Testing Overview
2. Pre-requisites for testing
3. Unit/Development Testing
4. System Testing
5. Total Code Coverage
6. User/Release Testing

Testing Overview

Testing of our software project took part in three phases, namely: the Unit testing phase, the System testing phase and the User/Release testing phase.

The unit testing phase consisted of testing each of the individual classes, which our project comprises of, with the exception of the GUI and one of its helper classes. The System testing phase is where the basic functionality of the software was tested including with the use of the GUI. In this phase, all testing is automated and all classes were essentially tested as a whole working component. The User/Release testing phase is where the full functionality of the software is completely tested with both expected and unexpected input to encounter any unexpected scenarios or bugs.

The total code coverage of all test cases is 76.4%. This occurs when all the test classes and the system test class are executed all at once. The code coverage, of course, differed between each testing phase and will be further analyzed during the discussion of each testing phase.

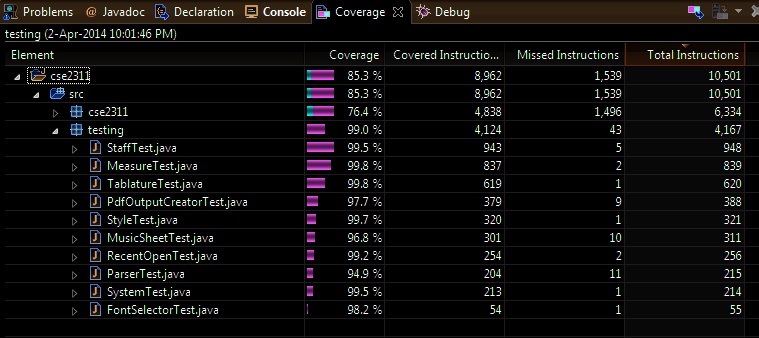


Figure 1: The percentage of code in the test classes that was executed

From figure 1, there are a total of 10 test classes, including the System test class. With these 10 test classes there is combined total of 72 test methods. Also apparent from the above figure is that a total of 4,838 instructions were covered out of a total of 6,334 instructions.

Pre-requisites for all Testing phases

The following is a list of the required resources used during all testing phases;

* The itext library
* A folder in the project directory called “test files”
* Six additional sub-folders in the folder “test files” named “measure”, “music sheet”, “Parser”,

“PdfOutputCreator”, “staff” and “tablature”. All sub-folders excluding the folders “Parser” and “PdfOutputCreator” should remain empty.

* Both the Parser and PdfOuputCreator contain special input text files which will be used in the testing phases. These input files will be discussed later on.
* The text file “rememberingrain” must be present in your default directory.
* The text tablature files: moonlight sonata, remembering rain, el Negrito and bohemian rhapsody.

Unit/Development Testing

This phase consisted of making individual JUNIT test classes for each class. There are a total of 9 test classes, with the UserInterface class and the PDFPanel class not having a specific test class. This testing phase ensured that all individual classes are all functioning individually properly.

The following 9 classes are unit tested: the Measure, Staff, Style, FontSelector, Parser, MusicSheet, Tablature, PdfOutputCreator, and RecentOpen. Only the testing of the important methods in each class are discussed and elaborated on.

Testing the Measure class

As the name implies this class deals with the things related to handling a measure like calculating the width, number of lines , the bar type and the number of times the measure is repeated. This class is tested by supplying in various inputs comparing the classes output to the expected values. The width is tested by supplying in input and comparing it the classes calculated width with the expected value. The bar type is tested by supplying all the possible bar types as input and then asserting that the class identified them correctly. The measure repetition was also tested similarly by supplying in input that needed to be repeated a different number of times and asserting that the class had obtained the correct value. The test class for Measure has coverage of 95.5%

Testing the Staff class

The staff class handles an array of measures. This class must handle the necessary information of the measures it contains. Methods of main importance in this class are the addToStaff( which adds measures), canFitAnother( determines whether another measure can be added ), and getTopInt( an array that stores the number of repetitions of each measure).

The addToStaff method is tested by continuously adding measures and then testing if all the measures are present and stored in the correct order. The canFitAnother method is tested by adding measures until it is determined theoretically that no more measures can be added after which it is asserted that no more measures can be added. The getTopInt is tested by adding measures that have different repetition values and then comparing the outputted values to the expected values. The test class for Staff has coverage of 100%

Testing the FontSelector class

The FontSelector class is responsible for returning a specific font type with a given index. This class is tested by selecting various font types and asserting that the class return the correct ones. This testing is sufficient because it ensures that the font corresponds to their correct index value. The test class for the FontSelector class has coverage of 91.1%

Testing the Tablature class

The tablature class handles an array of measures while it is being parsed from the text file. The main methods of importance in this class is the addMultiMeasureLine (which splits up multiple lines in a string and stores in a different measures), and the method addLineToLastMeasure (which adds a newly parsed single line measure to the last measure in the array list).

The testing of these methods consisted passing various inputs and checking if they performed their duties properly. For example when the addMultiMeasureLine encountered this

“||\*-----<5>-----<7>-----\*|| |-0-----7------0-------| ||\*-----<5>-----<7>----\*||”

It should create three new measures and divide the line into three and each instance should be placed in the three measures.

Similarly, the method addLineToLastMeasure should add the line “||\*-----<5>-----<7>-----\*|| ” to the last measure if and only if it is not full. If full then it makes a new measure instance.

The test class for the Tablature class had coverage of 80.4%

Testing the Style class

This class deals with the page layout and outputted tablature in the pdf. Since this class requires visually testing if the page layout indeed changed or not, the test class for Style only tests the logic and mathematical calculations of its inner workings.

The important method is getPrintSpace ( which determines the printable space allowed). This method is tested by passing in various values for the left margin and right margin and then comparing the outputted value to the theoretical value. The test class for the Style class has coverage of 97.4%

Testing the RecentOpen class

This class is one of the helper classes of the User Interface, which stores an array list of the most recently opened files. The logic behind this class is tested. Specifically whether the array list is being updated when the same file is opened twice and whether the size of the list is always 6.

This class is tested by adding the same files multiple times and then asserting that the class updated the list. It is also tested by adding more than 7 files and asserting that the size of the list is still 6 and that the oldest element is removed. The test class for the RecentOpen class has coverage of 81.2%

Testing the PDFPanel class

This is the second helper class of the User Interface which handles the live preview of the pdf. Due to the complexity of testing GUI components, this class does not have a Unit test but it is tested during the User/Release testing phase.

Testing the MusicSheet class

The musicSheet class creates new staffs from the given input of tablature with the information like printable space provide by the style class. To test this class, the method printStaff is of huge importance because once the tablature and style were passed, the only way to verify that the class worked would be through visually seeing the output.

Like a majority of methods that outputted to the console, the output was switched to a file called testPrintStaff, which is located in test files under MusicSheet. After the method is done outputting, the file is then opened and every line is compared to the expected output in the unit test class. For example, if the following measure is added twice in tablature,

|s4-----------------2-----| |-----4-----4-----4-----4-|

|-----4-----4-----4-----4-| |s4-----------------2-----|

|-------------------4---7-| |s4---1-------------------|

Then the musicSheet class should transform it into the following output in the testPrintStaff text file.

|s4-----------------2-----|-----4-----4-----4-----4-|

|-----4-----4-----4-----4-|s4-----------------2-----|

|-------------------4---7-|s4---1-------------------|

The test class for the MusicSheet class has coverage of 83.5%.

Testing the Parser class

The Parser class parses a text file and extracts the necessary information ( title, subtitle, spacing, measures, etc ) from it and returns it as a tablature. It also logs any errors and unusual stuff it encounters in the text file during the parsing process. There are 6 input text files that are used as testers. Once the parser was done parsing a file, the extract information was tested and compared with the expected output.

Some input files are meant to test the expected and unexpected input the user may pass in. The input text files can be found in the test files directory under Parser. Each input file and the necessary information that should have been extracted is provided below

* Empty file

An empty file is parsed. The parser sets the title, subtitle and spacing to its default values which are “Default”, “Default” and 5 respectively. The obtained values are asserted to equal the default values.

* ParserTest1

This text files only contains the following:

title= Remembering Rain

Subtitle= Jim Mateos

Spacing= 5.5

The obtained values are asserted to equal the above values.

* ParserTest2

This input text file contains perfect/ideal input that the parser class handles with ease and produces the correct expected output. There is absolutely no error outputted during this parsing process,

* ParserTest3

This text file contains only the following

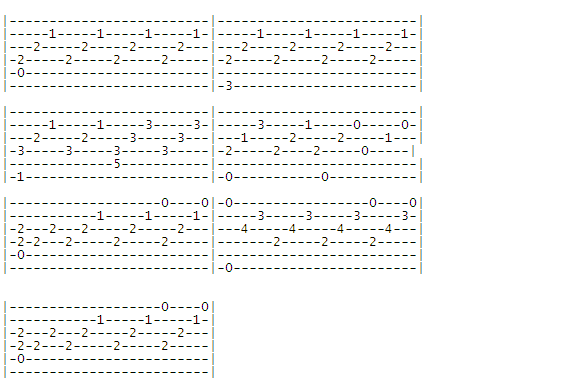


Figure 2.1: An input text file used to test the Parser Class. This file only contains measures and nothing else

The obtained information is then tested that it has been set to its default values, and the number of measures obtained are asserted to equal 7. There is absolutely no error outputted for this.

* ParserTest4

The first few lines of this text file have been provided below

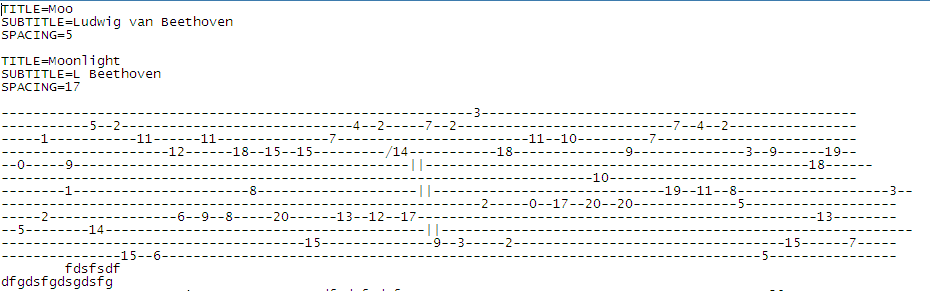


Figure 2.2: An input text file, which contains a lot of errors and two sets of title, subtitle and spacing

This input text file contains two sets of title, subtitle, and spacing. The parser only uses the last set, so the title, subtitle and spacing are asserted to be “Moonlight”, “L Beethoven” and 17 respectively. This text file outputs all the errors in this file to the output stream when it is done parsing. These errors are then visually compared to the expected errors. For example the first few errors should be outputted as follows:

INFO: Line 9 has no separating characters `|`:-----------------------------------------------------------3-----------------------------------------------

INFO: Line 10 has no separating characters `|`:-----------5--2-----------------------------4--2-----7--2---------------------------7--4--2----------------

INFO: Line 11 has no separating characters `|`:-----1-----------11------11--------------7------------------------11--10---------7-------------------------

INFO: Line 12 has no separating characters `|`:---------------------12------18--15--15---------/14-----------18--------------9--------------3--9------19--

INFO: Line 14 has no separating characters `|`:--------------------------------------------------------------------------10-------------------------------

* ParserTest5

This is the last input test file used and it contains semi-ideal input with the exception with a few indenting errors and missing characters.

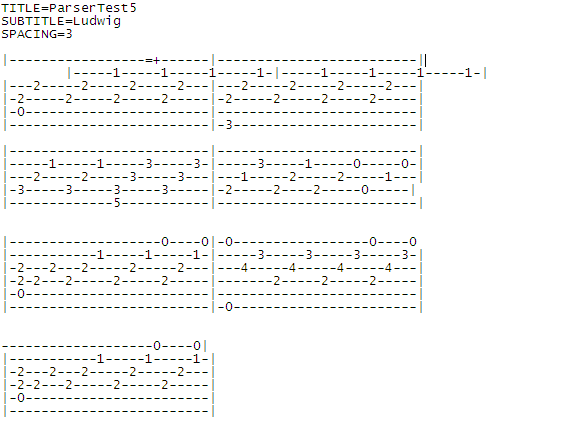


Figure 2.3: An input text file, that contains some improper indentation and a few missing characters

The only expected and outputted error logged is

INFO: Line 6 ignored: |-----1-----1-----1-----1-|-----1-----1-----1-----1-|

All other errors like missing characters and uneven measure lengths are automatically corrected and not logged.

Any errors that are present in the file are handled by the parser class either through logging or automatically fixing it. All the test cases used above are indicative of possible user inputs and also of inputs that may not consist of the accepted symbols of this software. These tests show how the program handles unexpected input. The test class for the Parser class has coverage of 89.8%,

Testing the PdfOutputCreator class

The PdfOutputCreator class converts a tablature to a PDF. The main method of importance in this class is the makePDF() method. This class is tested by creating pdfs for 6 text files and then visually checking if the pdfs were created and also if the necessary information were drawn in it.

This class was tested by first converting an empty file, a tablature containing only title and subtitle, a tablature containing only numbers, and a tablature containing advanced music notes and symbols. After the pdf file is created, the unit test checks if the file exists and extracts the text to check if the numbers and words are in the correct order and position.

* Empty file ( pdfOutput\_test1 )

Figure 2.4 shows the PDF output of an empty file. The title and subtitle of an empty file is automatically set to the string “Default”. For testing the pdf in figure 2.4, the text from the document is extracted and asserted to equal the string “Default\nDefault”. This technique is also used to test all the created PDFs.

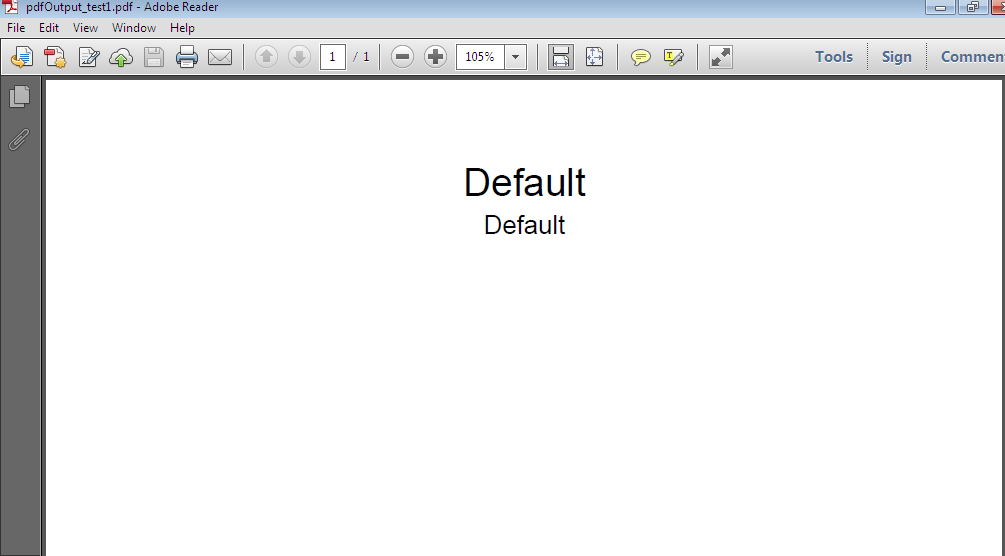


Figure 2.4: The PDF output of an empty text file

* pdfOutput\_test2 ( tablature with only title and subtitle )

The following is the content of this text file:

title=pdfOutput test2

Subtitle=Jim Mateos

Spacing=5.5

The output pdf should contain only a subtitle and title which are Jim Mateos and pdfOutput test2 respectively. Figure 2.5 illustrates what the outputted pdf should look like. This test case, more or less, tests whether the class is not outputting Default as the title and subtitle every time.

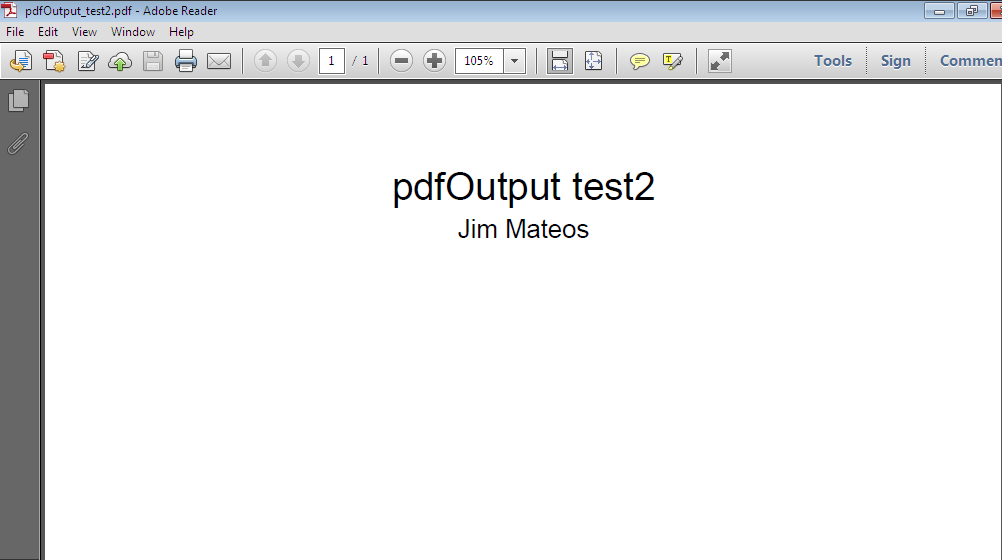


Figure 2.5: A PDF output of a file that contains only a title and subtitle

* pdfOutput\_test3 ( tablature containing only numbers)

This test case checks to see if the numbers in a tablature are printed in the pdf in the correct order, form and quantity. The test input file is displayed in figure 2.6.

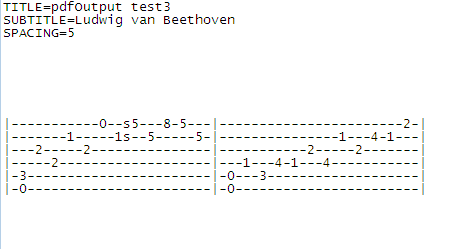


Figure 2.6: An input text file that contains numbers as its only symbol

Output file:

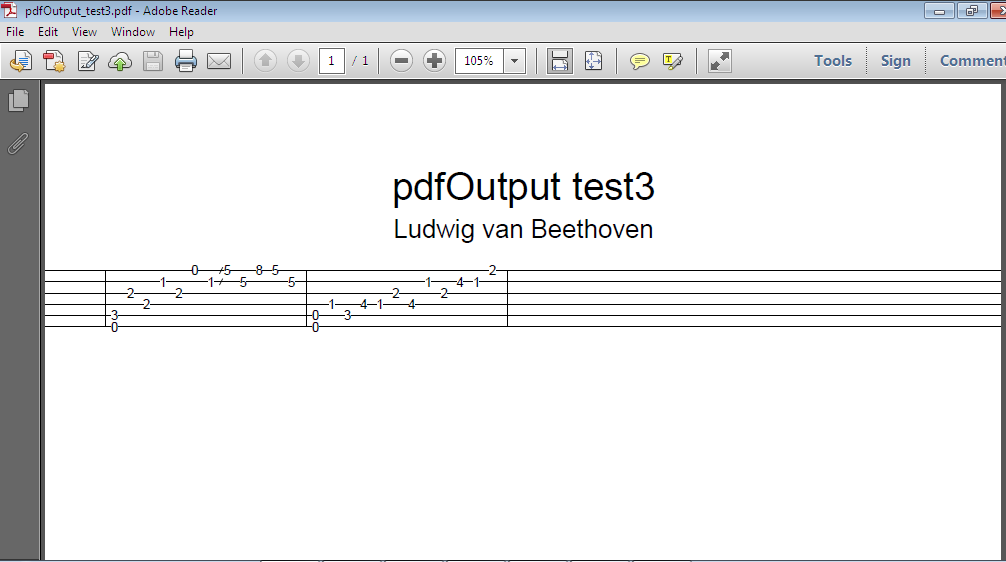


Figure 2.7: The PDF output of the text file in figure 2.

* Pdf\_Output\_test4 ( tablature containing advance symbols )

This test case, tests whether tablature symbols are being drawn in the pdf correctly in the right places, size and shape. This test case involved manually looking at the symbols to determine if everything was done correctly.

Input file:

TITLE=pdfOutput test4

SUBTITLE=Jim Mateos

SPACING=4.4

||------------------------------<12>----------||

||-<12>------------------------------<12>-----||

||\*-----<5>-----------<7>--------------------\*||

||\*---------------<7>------------------------\*||

||--------------------------------------------||

||--------------------------------------------||

||---------3----------10-----0-------0---------------7-|

||-----0---------10--------------0-------0-------5s7---|

||\*------2----------0----------2-------2-------0-------|

||\*--------------------------------2-------------------|

||---2---------------------3---------------------------|

||-0-----------7-----------------------------0---------|

|---7---------------------|-----------5---8-5-12 8---|

|-9-----9-----------------|-------5-----5------- --10|

|-----10--7-10--7---------|---5-----5----------- ----|

|-------------9----s6---0-|-7---7--------------- ----|

|-----------------8---7---|-7------------------- ----|

|-0-----------------0----0|-5-----------------0- ---0|

|---------7--------|----------5---------|

|-----5s7---7------|------3s5---5-----0-|

|---0---------0----|--------------0-----|

|----------------2-|----2-----------2---|

|------------------|-2------------------|

|-0----------------|--------------------|

|---7---------------------|-----------5---8-5-12 8---|

|-9-----9-----------------|-------5-----5------- --10|

|-----10--7-10--7---------|---5-----5----------- ----|

|-------------9----s6---0-|-7---7--------------- ----|

|-----------------8---7---|-7------------------- ----|

|-0-----------------0----0|-5-----------------0- ---0|

|-------0-----------------|-------------5-----------| |-------------------------|

|-----1---1---------------|-------------5-----------| |-------------------------|

|---2-------2---2---------|-------------5-----------| |-------------------------|

|-2-----------2---2---2---|-------------7-----------| |-7-----------------------|

|-0-----------------3---3-|p0-----------0-----------| |-0-----------------------|

|-------------------------|-------------------------| |-------------------------|

Produces the following output file:

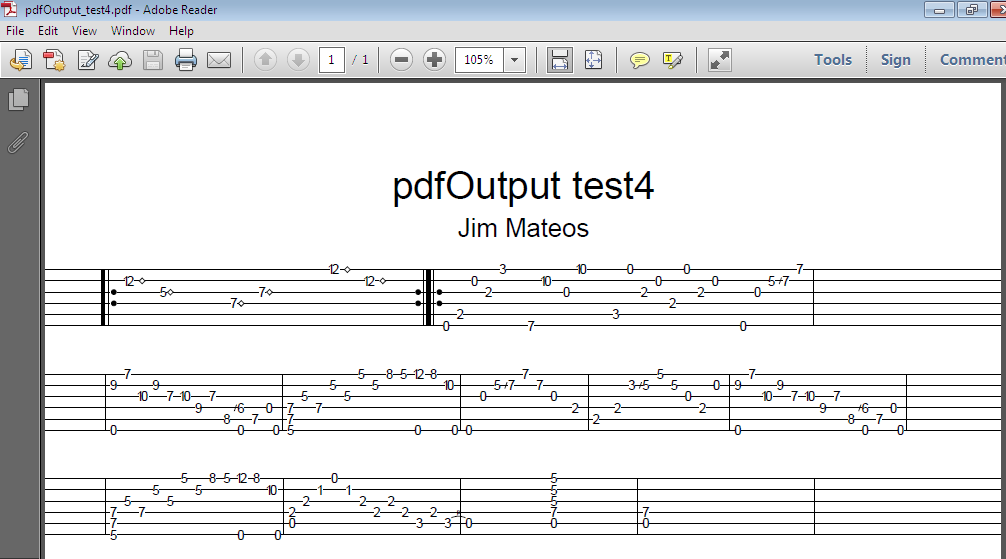


Figure 2.8: The PDF output of the file pdfOutput\_test4 which contains all the acceptable symbols of a tablature

Using the above input is sufficient testing for this class because it tests all possible input types and symbols, and how it will appear in the pdf. The test class for the PdfOutputCreator class has coverage of 92.3%

Code Coverage Statistics for the Unit Testing Phase

The total code coverage of the unit testing phase is 81.7%. This percentage is higher than the value stated in the overview because the class UserInterface is excluded from the package in which all the classes reside. The following is the coverage statistics for the unit test classes:

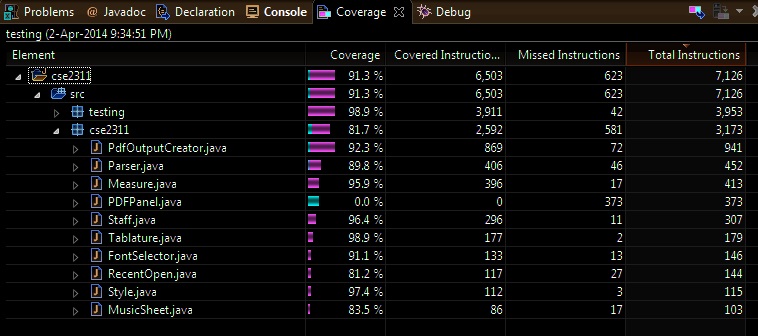


Figure 2.9: Code coverage statistics for all Unit tests, when executed all at once

As figure 2.9 shows, the JUNIT test classes cover a total of 2592 instructions out of a total of 3173 instructions.

The coverage percentage of main importance is the value listed beside “cse2311” because that is where the classes being tested reside. The package called “testing” contains all the unit test classes.

The coverage of each individual indicates that the JUNIT test classes covers most of the necessary code blocks. The class PDFPanel has 0% coverage because it’s one of the helper classes of the GUI and hence involves testing a graphical component which is extremely difficult. The testing of this class and the GUI is partially tested in the system test phase and fully tested in the user/release testing phase.

System Testing Phase

In this testing phase, a JUNIT test class was written so that it performs an automated test of the basic overall functionality of the software. In this phase, a file is converted to a pdf through the use of the GUI unlike how the pdf was generated in the Unit test phase. The test case serves as a basis to test every single class in the used in the software including the GUI and its helper class, PDFPanel. It also serves as a test to ensure that the GUI and the backend are properly in sync and that the basic functions (open and save ) of the GUI are correctly functioning.

This automation involves the use of the GUI to open the text file “rememberingrain” in the default directory and convert it to a pdf named “systemtest” in the project directory. Once the automation closes the GUI, the created file is then tested to assert that the file was indeed created, the size of the file is not 0, and the creation date of the file is equal to the exact minute of the current machine. Once the necessary tests are completed, the systemtest pdf is then deleted to allow the next execution of this class to encounter no problems.

The automation is performed with the java robot class which simulates keyboard and mouse inputs.

How it works

The following is a step by step of how the system test class is executed, along with snapshots of each phase.

1. In the first step of the system class, the GUI is started and the open button, by default, is selected so hitting enter will cause the file chooser to pop up.

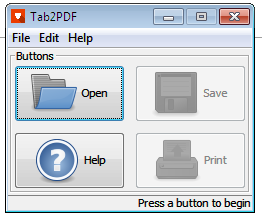


Figure 3.1: First step in the execution of the automated test class

1. Once the program hits enter, a file chooser will appear pop up and the program will type in the words “rememberingrain.txt” which should be located in your default directory.

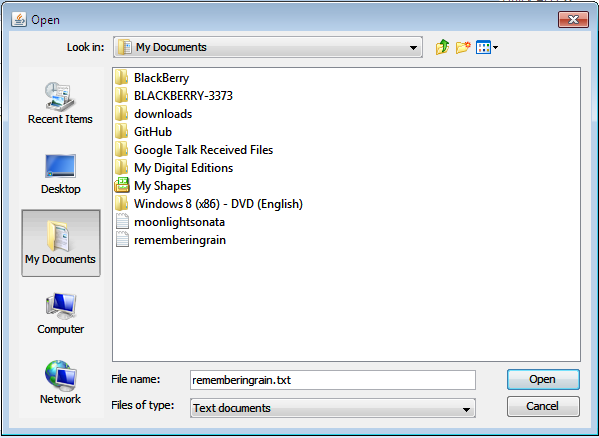


Figure 3.2: The automation then hits the open button and types in the file name rememberingrain.txt. It tests the opening function

1. The program then hits enter which opens the remembering rain text file. Once the file is opened successfully, the following is displayed.

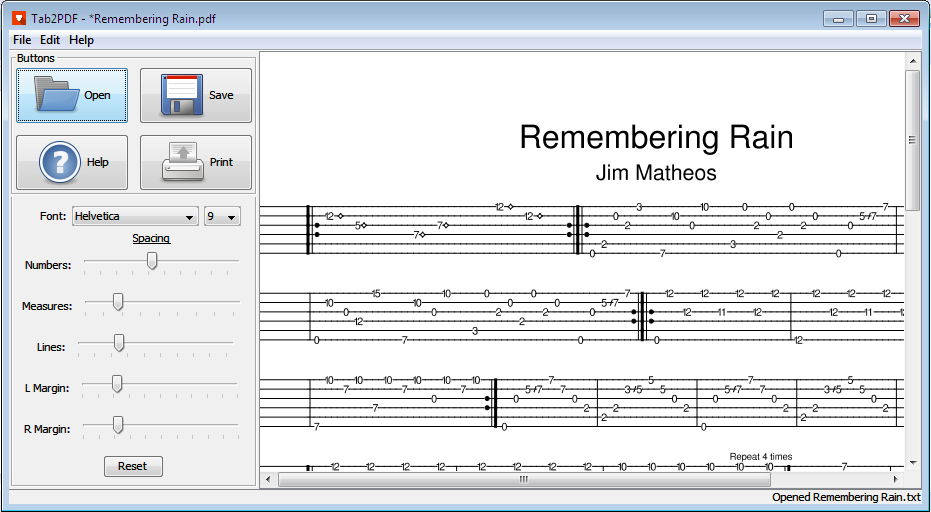


Figure 3.3: The view of the GUI when the file is opened. This tests the live preview function of the GUI

1. The program then hits tab, which selects the save button and hits enter. A save dialog box appears and the program types in the directory to the project folder and saves the file as systemtest.
2. After the pdf is saved the System tests performs the next set of tests on the newly created pdf.

As illustrated in steps 1 – 5, the system test covers the most fundamental actions a user will perform to convert a tablature to pdf using the GUI. This testing phase ensures that the program performs its primary function without encountering any unforeseeable error or hindrances like the file not being successfully created.

Code Coverage Statistics for the System Test Phase

The total code coverage of the system test class is exactly 69.0% as indicated in the below figure. For this coverage statistic the UserInterface class was added into the same package as the other classes, so the total numbers of instructions are greater than it was in the unit testing phase. A total of 4584 instructions were covered out of 6548, with almost half of these instructions coming from the UserInterface class.

The code coverage in the UserInterface ( GUI ) is 59.7%. Although this percentage isn’t great, it is quite reasonable since the GUI has more functionality than just opening and saving a file. The additional functionality is further tested in the user/release testing phase. PDFPanel, a helper class of the UserInterface), has a code coverage of 75.6% which indicates that a majority of its functions/code is covered.

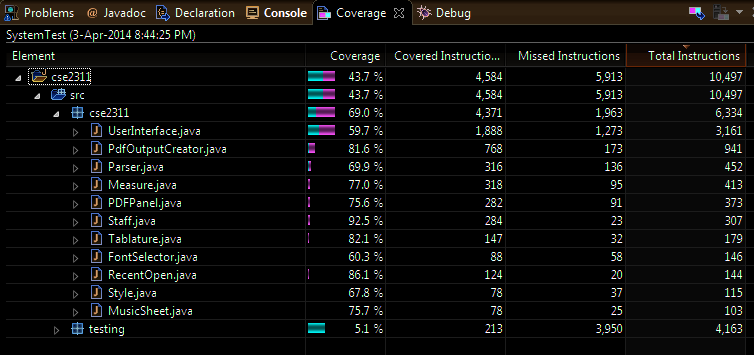


Figure 3.4: Code Coverage Statistics for the System test class

Coverage of the Unit and System Test Phases

As indicated in figure 4.1, the total code coverage obtained when both the unit tests and the system class are executed at the same time is 76.4% In terms of instructions, 4,838 out of a total of 6,334 instructions are covered. The UserInterface contains roughly about 50% of these instructions. Since the UserInterface has more functionality which cannot be tested with JUnit classes, the total code coverage of 76.4% is a very reasonable value.

If the instructions for the advanced functionality of the UserInterface were absent, the code coverage would definitely be higher than it currently is. Knowing this, the coverage of 74.6% is extremely good because it indicates that the overall software can perform its main functions.

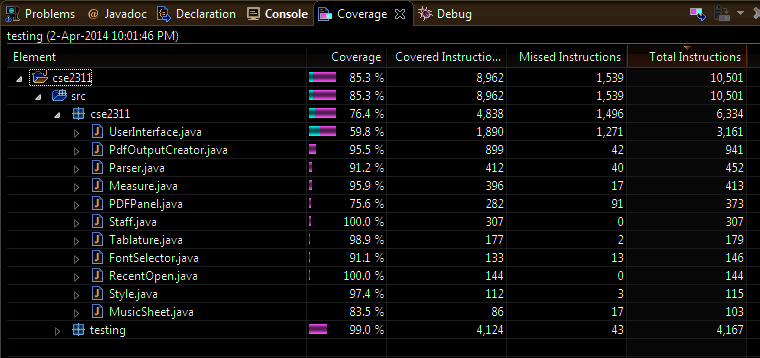


Figure 4.1 Code Coverage statistics when both the system test class and the unit test classes are executed at the same time

User/Release Testing

This phase consists of testing the overall full functionality of the software by converting numerous tablature files, both good and bad, and seeing how the program responds to it while fixing any encountered errors. In this phase the software is used in the same way the customer will use it.

The user/release testing phase was divided into two sub-phases: the acceptance testing phase and the advanced functionality testing phase. The acceptance phase consisted of comparing the output of our program, when given the tablatures remembering rain and moonlight sonata, to the pdf output the customer expects. The customizable testing phase involved creating pdfs that had their font and style changed,

Acceptance Testing

By using the two text files, specified by the customer, as inputs it was determined that the program produced PDF versions of them that were acceptable within the tolerated error. These two input files are remembering rain and moonlight sonata. If the outputted PDFs had failed to fall within the error tolerance, then the program would have failed the testing phase and the encountered bugs would be addressed immediately.

Acceptance case 1: Moonlight Sonata

Acceptable Output:

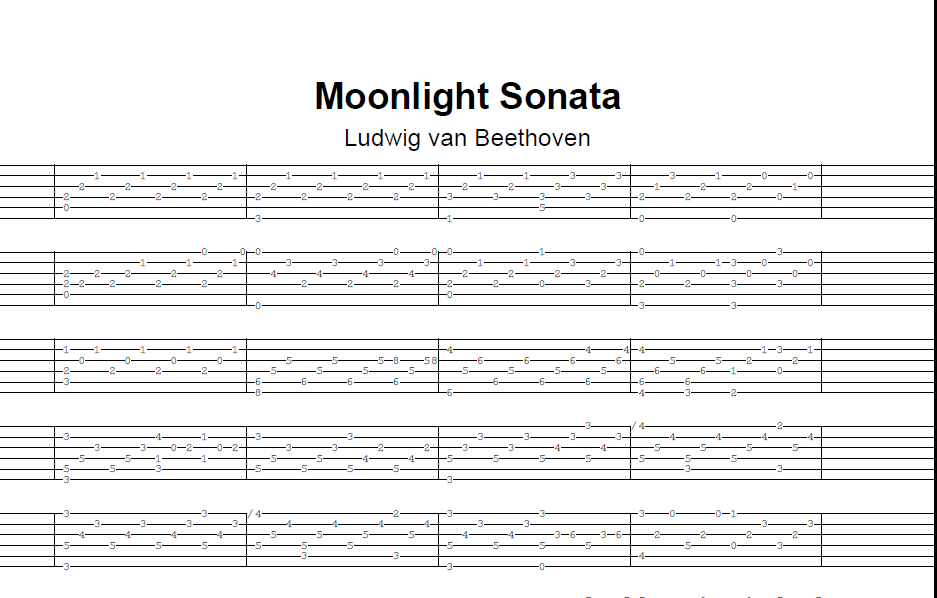


Figure 5.1: The expected PDF output of the tablature Moonlight Sonata

Program’s Output:

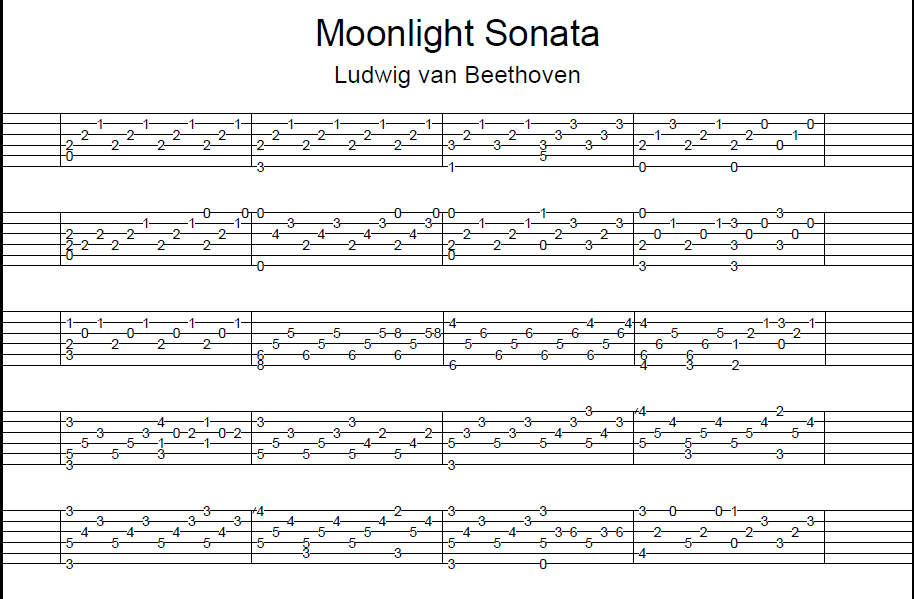


Figure 5.2: The software's PDF output of the tablature Moonlight Sonata

Acceptance Case 2: Remembering Rain

Acceptable Output:

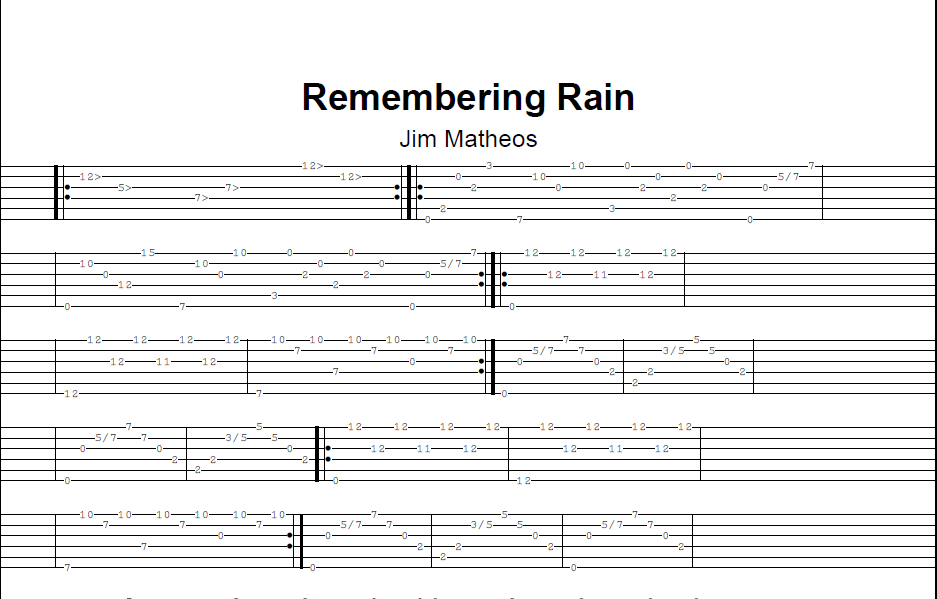


Figure 5.3: The expected PDF output of the tablature remembering rain

Program’s Output

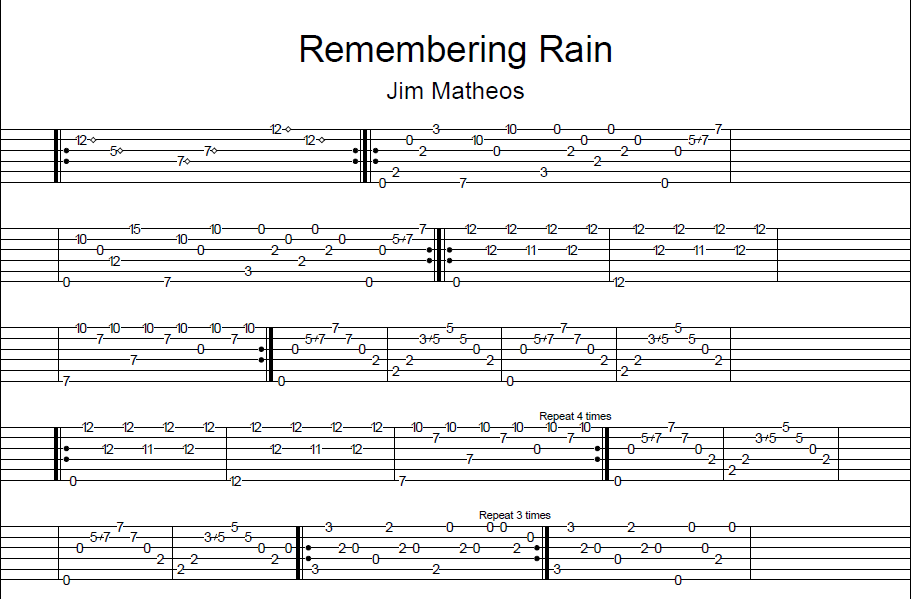


Figure 5.4: The program's output of the tablature remembering rain

Advanced Functionality Testing

Once the acceptance testing passed, the full functionality of the software was tested by converting various text files and add customizations the outputted pdf. In doing these customizations, the advanced functionality of the GUI, like the live preview is also tested thoroughly. The following first 3 inputs are tablatures that were found online and contain symbols and notations not defined in the requirements of the customer. Here are the tablatures and the software’s outputted pdf version of it:

* Bohemian Rhapsody Tablature

TITLE=Bohemian Rhapsody

SUBTITLE=Queen

Arranged by Edgar Cruz

Transcribed by Greg Palma

Gtr I (E A D G B E) - 'Untitled'

Intro

Q=76

A6 B7 A6B7F#B

4/4

Gtr I

E E E E Q Q E E E E E S S +Q

|---5-5-5-5---5---|---5-5-5-5-5--------|

|---7-7-7-7---7---|---7-7-7-7-7-7------|

|---9-9-9-9---9---|---8-8-9-8-6-8------|

|-----------------|-------------9------|

|-----------------|--------------------|

|-----------------|--------------------|

E7 D E7 A D A BmA

5/4

E E E E Q Q E E E E E E E E +H

|---4-4-4-5---4-------|-5-5-5-5-2-0-------|

|---5-5-5-7---5-------|-5-5-7-5-3-2-------|

|---7-7-7-7---7-------|-6-6-7-6-4-2-------|

|---------------------|-------------------|

|-----------------7-7-|-------------------|

|---------------------|-------------------|

F#m A7 G A7 A

4/4

E E E E E E E E E E E E +E E +E E

|-----------2-----|-------0---------|

|-2-2-2-2-----2---|-2-2-3-----------|

|-2-2-2-2-------2-|-0-0-------2---7-|

|-----------------|-----------------|

|-----------------|-0---------------|

|-2-2-2-2-2-------|-----------------|

D G D Bm7 Bm9

E E E E E E E E E E E E +E E Q

|-------5---10--10-|-----------9-----|

|-----7---12--10---|-7-7-7-7---------|

|---7-----12--11---|-7-7-7-7---------|

|-0----------------|-7-7-7-7---------|

|------------------|-----------------|

|------------------|-7---------------|

W

The PDF output of the above is

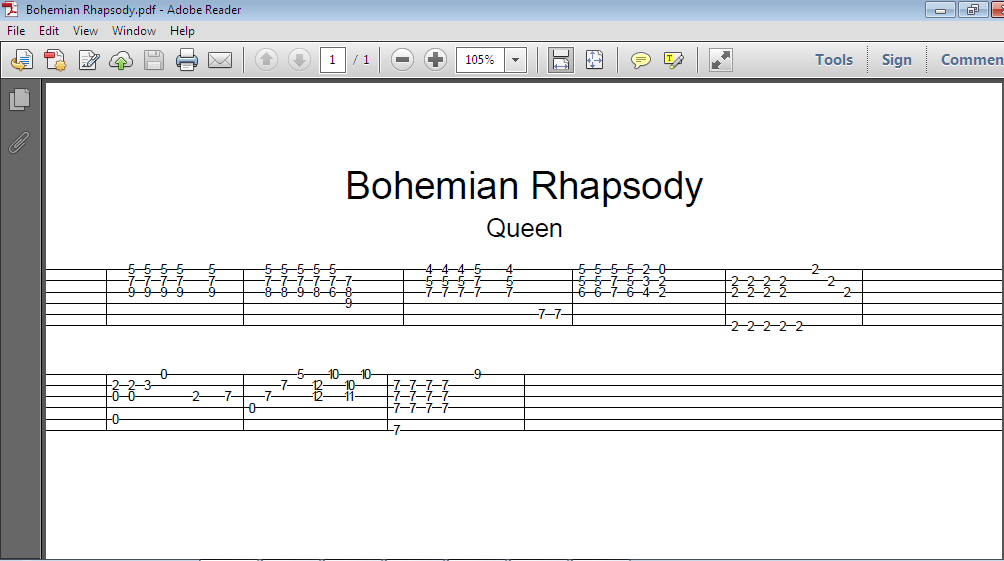


Figure 5.5: The pdf output of the tablature bohemian rhapsody which contains some unaccepted symbols and notations

The produced output is reasonable since some of the symbols in the input were not defined in the alphabet of acceptable strings defined by the customer. The software ignores any lines that contain foreign symbols and prints the correct ones.

* El Negrito Tablature

TITLE=El Negrito

SUBTITLE=Antonio Lauro

El Negrito (from Two Venezuelan Waltzes) - Antonio Lauro (1917-1986)

This is one of the loveliest guitar pieces, and it's easy to play!

Key C

Time 3/4

Tempo 120-132

SPACING=6.0

0 | | | | | | | | | | | |

|---0-1-2-5-3-||-3-----------|---------1-0-|-0-----------|

|-------------||---------3---|-----1-3-----|-----------0-|

|-------------||\*----0-----0-|-----2-------|---------0---|

|-------------||\*--0---0-----|---2---------|-----2-4-----|

|-------------||-2-----------|-0-----------|---2---------|

|-------------||-------------|-------------|-3-----------|

        1-1 4 2 4 3 2 1 4 1 2 3-3

                    1 3 4

4 | | | | | | | | | | | |

|-------------|-------------|-------------|-0-----------|

|-----------0-|-1---------1-|-3-------0-3-|---1---1-----|

|---------2---|---------0---|-------0-----|-----0-----0-|

|-----0-3-----|-------2-----|-----3-------|---------2---|

|---3---------|-----3-------|---2---------|-3-----------|

|-1-----------|-0-3---------|-3-----------|-------------|

      3 4 2 1 4 2 1 4 1 3 4 1 2

    1 3 2 3

8 | | | | | | | | | | | |

|---0-1-2-5-3-|-3-----------|---------1-0-|-0-----------|

|-------------|---------3---|-----1-3-----|-----------0-|

|-------------|-----0-----0-|-----2-------|---------0---|

|-------------|---0---0-----|---2---------|-----2-4-----|

|-3-----------|-2-----------|-0-----------|---2---------|

|-------------|-------------|-------------|-3-----------|

12 | | | | | | | | | | | |

|-------------|-------------|---------0---|---0---------|

|-----------0-|-1---------1-|-3---------3-|-1-----1-----|

|---------2---|---------0---|-------0-----|-----0-----0-|

|-----0-3-----|-------2-----|-----3-------|---------2---|

|---3---------|-----3-------|---2---------|-3-----------|

|-1-----------|-0-3---------|-3-----------|-------------|

Output Pdf



Figure 5. 6: The pdf output of the tablature el negrito. This tablature contains some unaccepted notations

* Unlikely User input

TITLE=Moo

SUBTITLE=Ludwig van Beethoven

SPACING=5

TITLE=Moonlight

SUBTITLE=L Beethoven

SPACING=17

-----------------------------------------------------------3----------------------------------------------

-----------5--2-----------------------------4--2-----7--2---------------------------7--4--2---------------

-----1-----------11------11--------------7------------------------11--10---------7------------------------

---------------------12------18--15--15---------/14-----------18--------------9--------------3--9------19-

--0-----9------------------------------------------||------------------------------------------------18---

--------------------------------------------------------------------------10------------------------------

--------1----------------------8--------------------||-----------------------------19--11--8-----------3--

-----2----------------6--9--8-----20------13--12--17--------------------------------------------------13--

--5--------14----------------------------------------||---------------------------------------------------

----------------------------------15--------------9--3-----2----------------------------------15------7---

---------------15--6---------------------------------------------------------------------------5----------

fdsfsdf

dfgdsfgdsgdsfg

Output pdf

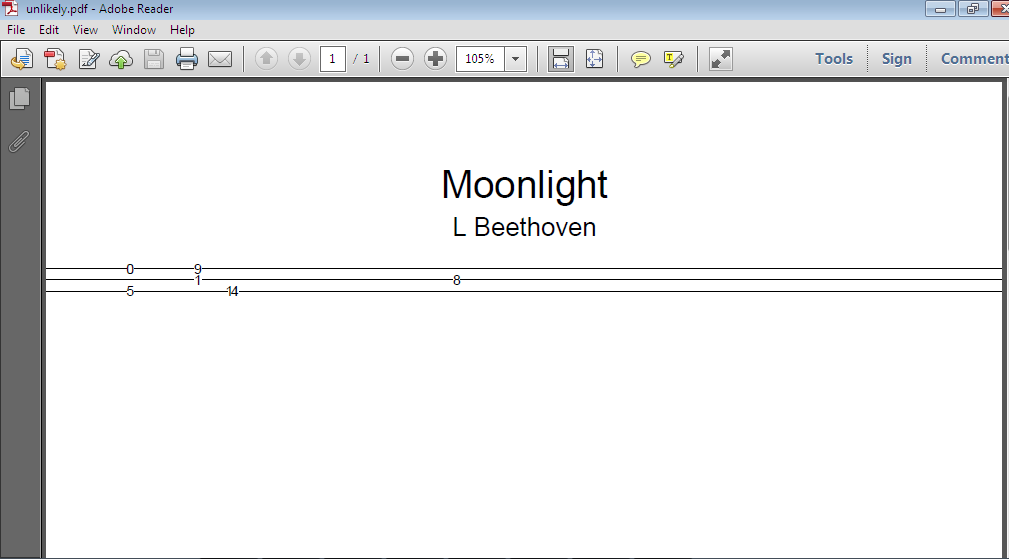


Figure 5.7: The PDF output of the above tablature text file. The program was only able to print the lines that made sense to it

The customizable option of the software is also tested by applying various changes to the remembering rain tablature and comparing it to the original version to determine if the change took effect.

Testing of customizable features

In the testing of the advanced features, everything was done visually and with the use of the GUI and its live preview, to enable full test coverage. The customizable features that were fully tested are font type, measure distance, spacing and line spacing. All other customizable features not mentioned below were also fully tested visually too.

The following are the various types of pdf, which can be created from the same tablature, which can be created with this software. These multiple pdfs serves as results of testing the customizable features of the software.

* Remembering rain ( Original version )



Figure 5.8: The program's default PDF output of the tablature remembering rain

* Remembering rain with a different font type

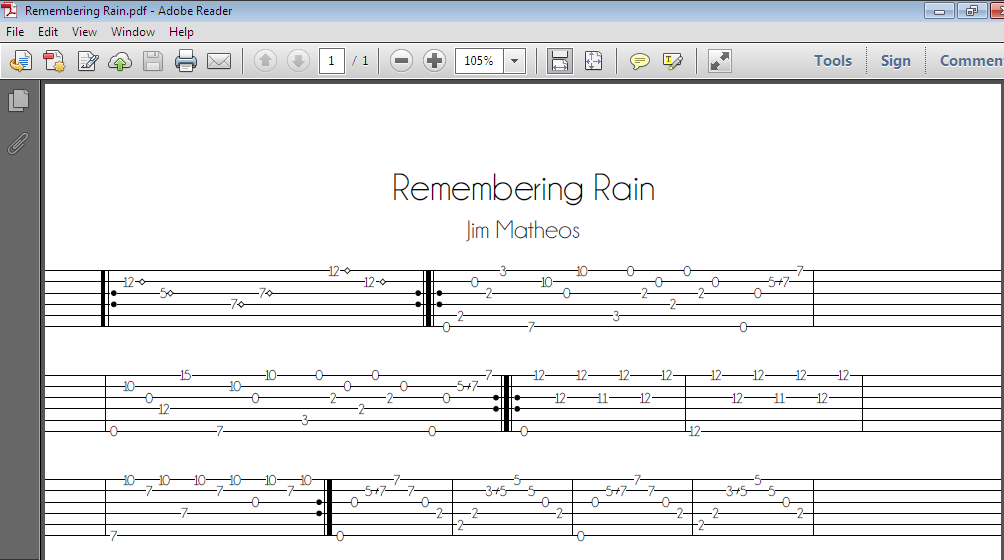


Figure 5.8: The program's PDF output of the tablature remembering rain with a different font style

* Remembering rain with a different spacing value

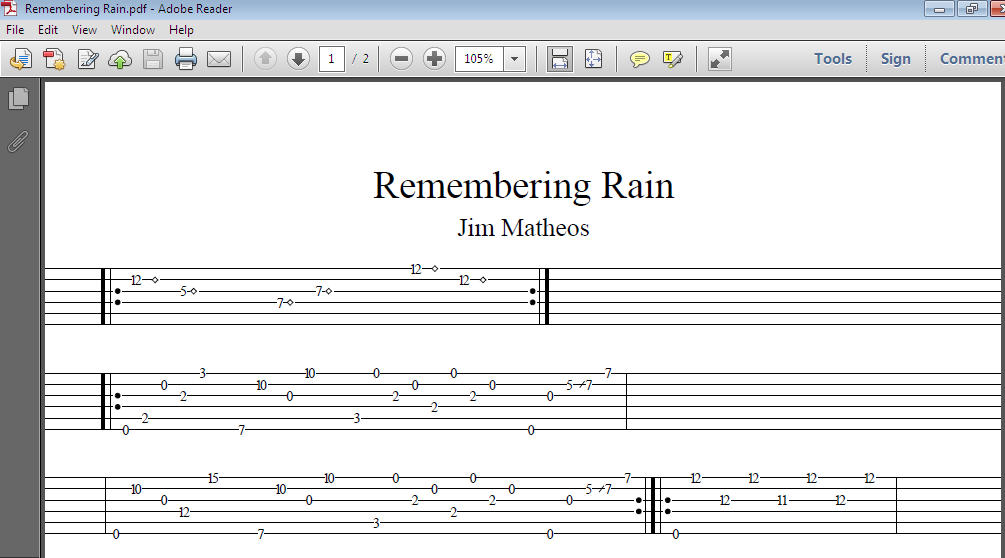


Figure 5.9: The PDF output of the tablature remembering rain with a different spacing value

* Remembering rain with a different font type, different spacing value, different measure difference and a different line spacing.

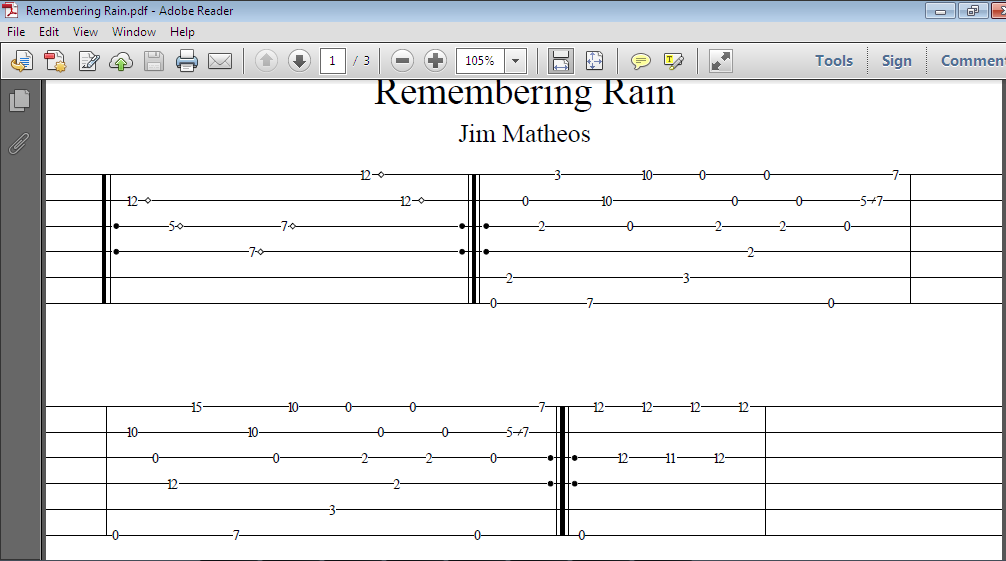


Figure 5.10: The PDF output of the tablature remembering rain with a set value for the measure distance and the line distance

Coverage of the User/Release Testing Phase

Although this testing phase can’t be statistically analyzed with numbers, it is certified that this software will be able to dynamically handle any kind of input without crashing or malfunctioning. This conclusion is evident from the fact that it was able to handle a wide range inputs with efficiency and ease.

The full functionality of the GUI was also tested in this phase, and the estimated coverage of the instructions in the UserInterface class is 96% since all the functionality was fully tested and employed to customize the created PDF.