TabToPDF

Design Document

Group 1

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

CSE2311 Prof. Tzerpos

TABLE OF CONTENTS

1. INTRODUCTION 3
1.1 Purpose 3
1.2 Scope 3
1.3 Overview 3
1.4 Reference Material 3
1.5 Definitions and Acronyms 3
2. SYSTEM OVERVIEW 3
3. SYSTEM ARCHITECTURE 4
3.1 Architectural Design 4
3.2 Decomposition Description 6
3.3 Design Rationale 7
4. COMPONENT DESIGN 7
4.1 Formatter 7
4.2 Conversion 14
4.3 GUI 16
4.4 Additional Features 19
5. MAINTENANCE SCENARIOS 21

1. INTRODUCTION

1.1 Purpose

The purpose of this document is to describe the major design and architecture, features, and the decisions that were made in developing the TabToPDF software.

1.2 Scope

The main purpose of this software is to convert a text file containing music information into tablature sheet music. The text file contains guitar fingering that follows a certain format. The resulting output file is in PDF format and resembles sheet music. This program will benefit a guitar musician of any level with a clearer tablature music sheet. Being versatile and lenient to errors, it is able to produce a professional looking sheet of music given a simple text file containing tablature. This software will also allow for printing, emailing, formatting and file management of the input and output files.

1.3 Overview

This document begins by introducing the software and necessary jargon to understand the software design and implementation. It is followed by the major codes responsible for conversion of the input files to its output will be discussed. The GUI will be described next. Then features such as printing, emailing, formatting and file management will be analyzed.

1.4 Reference Material

Please refer to the Test, Requirements and User Manual Documents.

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. SYSTEM OVERVIEW

The user runs the program and first has to choose an input text file that follows a required format. The user can choose from a number of files and choose the file name. The user then can convert the text file into PDF format after viewing a preview and/or formatting it by changing font, font size, etc. Once the PDF has been successfully generated, the user may email or print the document. This software is intended for guitar music and can be for the amateur musician wanting to create their own music or a larger group of people needing to create large quantities of sheet music.

3. System Architecture

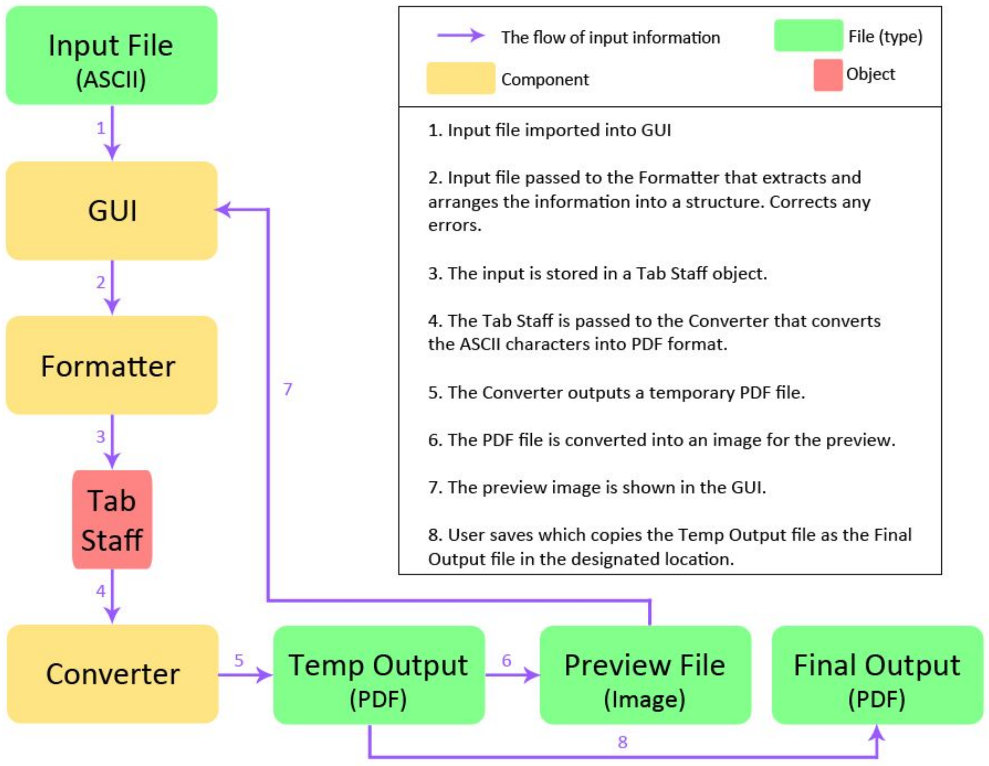
3.1 Architectural Design Components

The system consists of the following components:

Conversion
GUI
Emailer
Print
Formatter
Testing

The GUI coordinates the activities of Conversion, Email, Print and Formatter functions to deliver a high quality PDF document to intended recipients.

Selecting a File



1. Input file imported into GUI
2. Input file passed to the Formatter that extracts and arranges the information into a structure. Corrects any errors.
3. The input is stored in a Tab Staff object.
4. The Tab Staff is passed to the Converter that converts the ASCII characters into PDF format.
5. The Converter outputs a temporary PDF file.
6. The PDF file is converted into an image for the preview.
7. The preview image is shown in the GUI.
8. User saves which copies the Temp Output file as the Final Output file in the designated location.

Figure 1: Process of selecting an input file and converting it to a PDF.

Editing the PDF

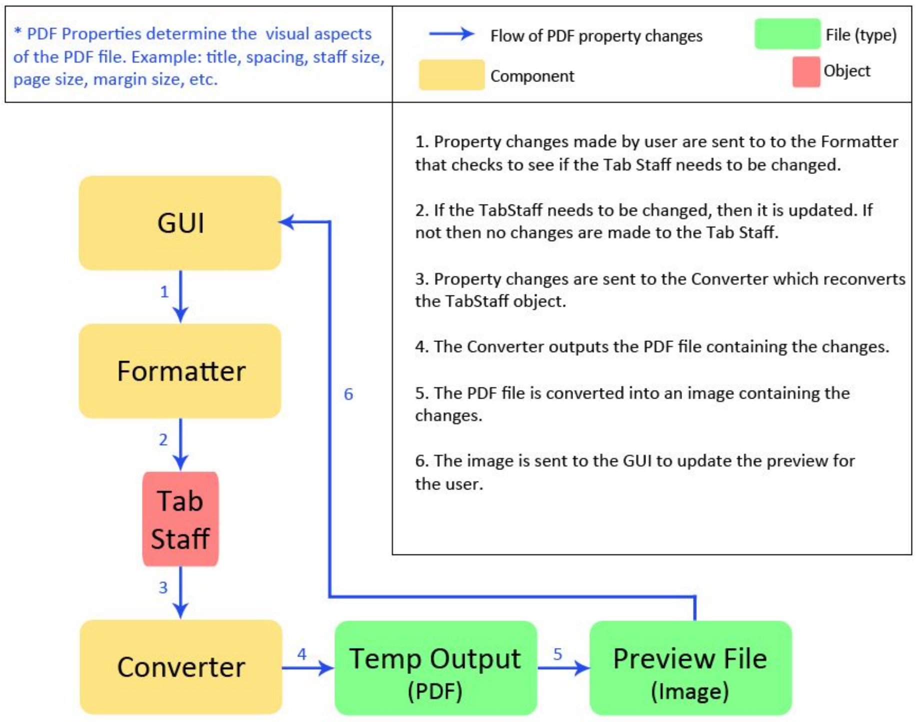

* PDF Properties determine the visual aspects of the PDF file. Example: title, spacing, staff size, page size, margin size, etc.
Flow of PDF property changes
Component
File (type)
Object
1. Property changes made by user are sent to to the Formatter that checks to see if the Tab Staff needs to be changed.
2. If the TabStaff needs to be changed, then it is updated. If not then no changes are made to the Tab Staff.
3. Property changes are sent to the Converter which reconverts the TabStaff object.
4. The Converter outputs the PDF file containing the changes.
5. The PDF file is converted into an image containing the changes.
6. The image is sent to the GUI to update the preview for the user.
GUI
1
Formatter
2
Tab Staff
3
Converter
4
Temp Output (PDF)
5
Preview File (Image)
6

Figure 2: Process of editing the PDF file.

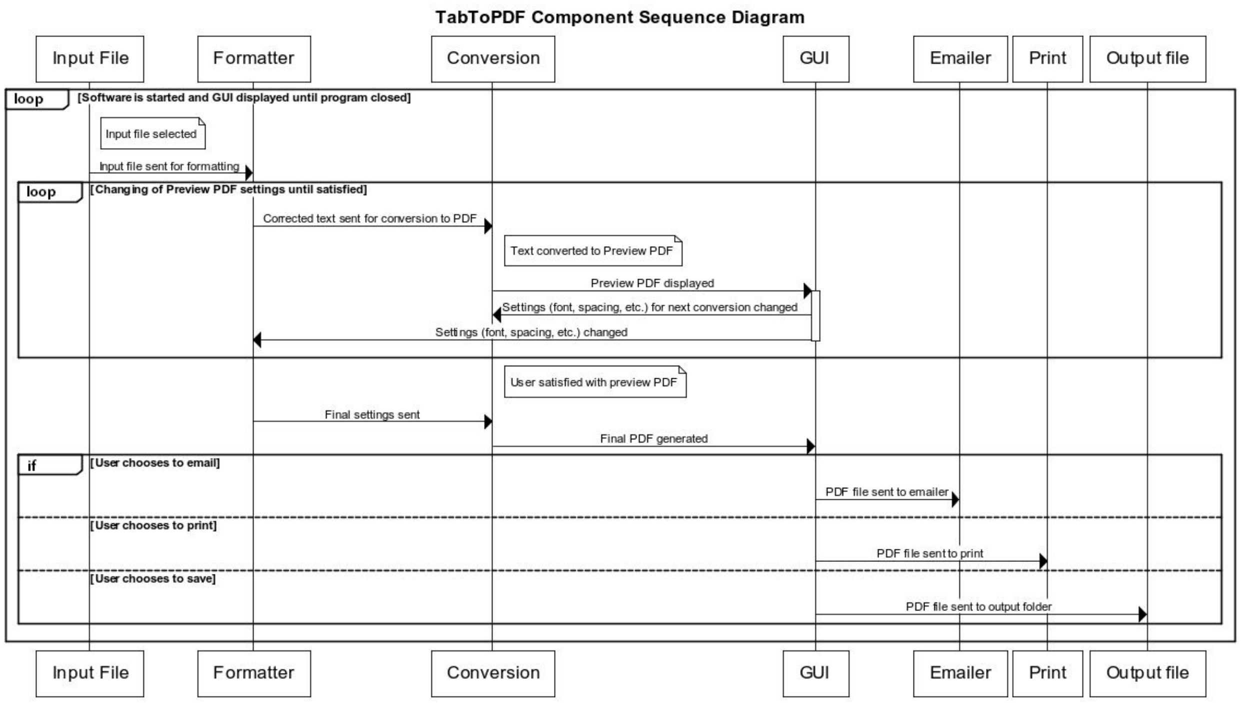

TabToPDF Component Sequence Diagram
Input File
Formatter
Conversion
GUI
Emailer
Print
Output file
loop
[Software is started and GUI displayed until program closed]
Input file selected
Input file sent for formatting
loop
[Changing of Preview PDF settings until satisfied]
Corrected text sent for conversion to PDF
Text converted to Preview PDF
Preview PDF displayed
Settings (font, spacing, etc.) for next conversion changed
Settings (font, spacing, etc.) changed
User satisfied with preview PDF
Final settings sent
Final PDF generated
if
[User chooses to email]
PDF file sent to emailer
[User chooses to print]
PDF file sent to print
[User chooses to save]
PDF file sent to output folder

Figure 3: General architecture overview of TabToPDF software. Arrows indicate data flow. Boxes indicate a programmed component that consists of multiple classes. Testing is its own component and does not interact with the software. Input Text File is supplied/created by the user or can be a sample that came with the software package. The Output PDF File is what the software delivers to the user.

3.2 Component Decomposition Description

Conversion – This component converts the corrected input file supplied by the Formatter into the output PDF file. It stores the data from the input file (.txt) in a multi-dimensional array which is used to create the PDF document.

GUI – This component controls all other components. The MVC design manages all the components with the View being the GUI, the Model creating an instance of all components and the Controller being the initiation of the program and the coordinator between the View and Model.

Emailer – This component is a feature that allows an individual to send the PDF through their gmail account to any other email account (including their own).

Print – This component searches for connected printers and allows the user to print their PDF.

Formatter – This component corrects errors in the input file so that Conversion can successfully generate the PDF. Errors include incorrect characters, formatting, spacing and more.

Testing – This component uses JUnit to test all components except the GUI.

3.3 Design Rationale

The components were chosen this way because of the key function they perform. The main idea is to compartmentalize each main function for efficient distribution to engineers/programmers. The largest function is converting the text input file into the PDF, which was designated by Conversion. The other functions are designated by the component names.

4. COMPONENT DESIGN

The components are shown as a group of classes that contribute to an overall function designated by the component name.

4.1 Formatter

Purpose

The purpose of the formatter is to prepare the input information before conversion. Putting the reading of the input data in a separate component from the converter allows for easier code maintainability and error handling. The converter doesn't have to worry about handling errors in the input because the formatter does that job already. The converter always runs confidently knowing it can draw the music staff without having to correct the input data. Bugs in the code are easier to fix because there is a distinction between a formatter bug and a converter bug. These are advantages from a coder's standpoint.

The primary goal in terms of what the user wants is to be able to read any ASCII file and produce a reasonable output, maintaining as much of the input tablature as possible, while not throwing errors that would frustrate the user. Therefore the formatter is designed to "think" for itself about what the music should look like given any type of input. It makes assumptions about what the user really wanted his music to look like if the input, for example, has missing staff bars, inconsistent string lengths or invalid music symbols.

In summary, the formatter has two main goals: to allow the converter to write the PDF without it worrying about tablature errors and to fix errors to produce a desired output for the user.

Valid Tab String

Before reading from a file, a definition for what music should look like needs to be defined. If an example of a valid tab string is "|----2p1--3----2|", we can derive that a string is valid if there are two “|” at either end of a set of characters containing dashes and other characters. The base case for a valid string then being “|-|” and “||-||”, “||-|” and “|-||” for the case of double bars. Therefore the general pattern for finding a valid string in word form is:

> Any set of characters where there are 1 or 2 bars at both ends with at least 1 dash and 0 or more characters in between the bars.

Of course not every character between the bars is acceptable but correcting that is done later. The point here is to use a loose definition of a music string to capture as much of the user’s tablature as possible, a definition that can be broadened even more.

Acceptable Tab String

What happens when the formatter encounters a line that matches the valid string pattern perfectly except that it’s missing one bar at the end? It would be bad design to completely ignore the line since most of it is music that the user is expecting to be output. The best solution is to create partially valid string patterns, the base cases being “-|” and“|-”. Then the definition of an acceptable string pattern is:

> Any set of characters where there are at least 1 or 2 bars at either end with at least 1 dash and 0 or characters before or after the bars.

This pattern would accept anything like “-----12---3p3--|” or “||----2---<1>”. We then use the **acceptable string pattern** to find lines in an input file that can be corrected into a **valid string pattern**. It’s important to note that the valid string pattern is a subset of an acceptable string pattern, and so the program only needs to search for acceptable string patterns when scanning a file.

Comments

What if there are no vertical bars? The pattern of an acceptable string has to stop broadening at some point otherwise it would accept any line of characters which isn’t very practical, especially if the user puts comments in the input file. A comment then is defined as anything that doesn’t match the acceptable string pattern and is therefore not converted into a music staff.

Class Overview

In order to send the corrected input to the converter, the formatter needs to organize the input data into some kind of structure. The main object classes of the formatter are TabString, TabMeasure and TabStaff. If the input looks like this:

```
|-------------------------|-------------------------|
|-----1-----1-----1-----1-|-----1-----1-----1-----1-|
|---2-----2-----2-----2---|---2-----2-----2-----2---|
|-2-----2-----2-----2-----|-2-----2-----2-----2-----|
|-0-----------------------|-------------------------|
|-------------------------|-3-----------------------|
```

A TabString object would contain one string: `|-----1-----1-----1-----1-|`

A TabMeasure object would contain one measure (6 strings):

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

Finally, a TabStaff would contain all the measures in the input file:

```
|-------------------------|      |-------------------------|
|-----1-----1-----1-----1-|      |-----1-----1-----1-----1-|
|---2-----2-----2-----2---|  and |---2-----2-----2-----2---|
|-2-----2-----2-----2-----|      |-2-----2-----2-----2-----|
|-0-----------------------|      |-------------------------|
|-------------------------|      |-3-----------------------|
```

Therefore, it makes sense to design the classes using composition. TabStaff would be composed of TabMeasures, and a TabMeasure would be composed of TabStrings.

The advantage of this becomes quite clear when trying to correct errors in the music. The TabString class would have its own fixString() method to remove any invalid symbols or add missing bars. The TabMeasure object would simply call the fixString() method for each of its 6 strings. In addition, the TabMeasure has its own fixMeasure() method which would fix errors in the music that fixString() can't such as unequal string lengths. Finally, the TabStaff would simply call fixMeasure() for all of its measures and in addition correct things that fixMeasure() can't like removing comments.

To word it another way, each object class makes corrections based on how much of the music it can "see". A class does not try to fix what is out of its reach to fix. For example, if the input looks like this:

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

This is a comment!

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

A TabString containing the first string of the first measure would know to remove the A's because they are invalid music symbols, but it doesn't know that its length is shorter compared to the other 5 strings. A TabMeasure object containing the first measure would see this error and be able to align the strings. The TabMeasure does not know to delete the "This is a comment!" line and so that job is handled by the TabStaff object which can see everything in the input file. The TabStaff class is not worried about fixing invalid symbols or aligning measures because the TabString and TabMeasure objects do those jobs for it.

This kind of composition makes it simpler to edit the code because, for example, if the definition of what a valid tab string needs to be changed, only the TabString class needs to be edited. The change transitions smoothly into the other classes. It also makes it easier to detect and fix bugs because a coder can easily look back at each TabString in each TabMeasure of the TabStaff to find the problem rather than rereading the input file one character a time.

TabString Class

The object uses a character array to hold the representation of a music string. It contains the string regex that are used for recognizing valid and acceptable string patterns, which are equivalent to the previous word definitions.

A key method of this class is the scanLine() method which reads one line and stores the first acceptable string it finds inside of its character array. It reads from one end of vertical bars to another which means it is not possible to store a vertical bar in the middle of the music string. Having this process cuts down on the amount of string regex that need to be used, those of which can cause long running times when called too often. For example scanLine() accepts the following line:

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

The object simply stores `---1----1!!!!1----1-|` inside of its character array and returns the starting index of the adjacent string.

The fixString() and fixSymbols() methods would add the starting bar to the above example and delete the exclamation points. Though it was previously mentioned that only one method handles the fixing of the string, in reality TabMeasure needs to call fixes for its TabStrings in a certain order to prevent correction conflicts (when one correction overwrites a previous correction).

The fixSymbols() method deletes special music characters like pull(p), hammer(h) and harmonics (<#>) if they aren't implemented correctly. For example, the following is the input:

```
|---0---<<-0-<->>---|
```

The method would remove all of the arrows since they don't match a valid harmonic pattern (<#>).

A notable attribute of this class is LogAttributes object which is used when storing data in the auto-correct log file (used to show what the program auto-corrected in the input). The LogAttribute stores what the TabString looked like before it was corrected so that the user can see a before and after comparison for all of the corrected strings.

TabMeasure Class

The object is composed of 6 TabStrings but it's considered a hybrid object because it can resemble a comment as well. This was done to preserve the order in which measures and comments are read from the input file and stored in the TabStaff object. It was originally planned to output comments along with the music but it felt superfluous in the end. The output should only print music. In any case, the hybrid functionality is still used since the TabStaff object easily deletes measures flagged as comments.

Notable methods in the class are equalizeStrings(), fixStartBar() and setRepeat(). The equalizeStrings() will add dashes or remove dashes from its strings to properly align the measure based on its maximum length. It only deletes dashes from strings that are blank (contain no number or symbols, only dashes) so the measure is aligned with the longest string that isn't blank.

The fixStartBar() will add bars to the start of each of its strings if they are missing. If there is a double bar at the start of any of its strings then it assumes that all of its strings should start with double bars. The same idea applies to the fixEndBar() method.

The setRepeat() method stores the repeat number found after the end bar of a measure. When the converter needs to write the statement “Repeats 3 times” above a measure, it simply retrieves the TabMeasure’s repeat number, which is stored as an attribute.

TabStaff Class

The object is composed of an array of TabMeasures. It uses the scanFile() method to read an input file. For every music string it detects, it creates a TabString which gets stored in a TabMeasure. Once all 6 strings are found, the TabMeasure is stored in the array of TabMeasures. If there is a line break before all 6 strings are found then empty TabStrings are added for missing strings. The method is designed to read and store all of the information *before* trying to fix anything. This is because there are corrections made to a TabString that only happen after its TabMeasure is established. The same goes for a TabMeasure and the TabStaff it belongs to. Another important reason is to be able to differentiate a TabStaff before fixes and a TabStaff after fixes so the code can be debugged easily.

The object uses the fixStaff() method which calls the fixMeasure() method for each of its TabMeasures in the array. It calls the deleteComments() method to delete the TabMeasures marked as comments. Finally, it logs the corrections made to a writeAutoFixLog object.

The method splitMeasures() is called when the length of a measure exceeds the PDF document bounds. In this case, measures exceeding the length are split in two, the second half moving to the next row. If the changes are undone, the measures are unsplit but instead recombining each string, the state of the TabStaff *before* it was split is reloaded. This is much more efficient, allowing the program to run faster.

Dependency

Unlike the converter component which relies on the formatter to produce a proper output, the formatter is highly independent from the rest of the program. All it needs is an input file and it produces a TabStaff object that contains the correct input. The advantage to this is that the formatter can be tested independently from the other components, making it easier to debug.

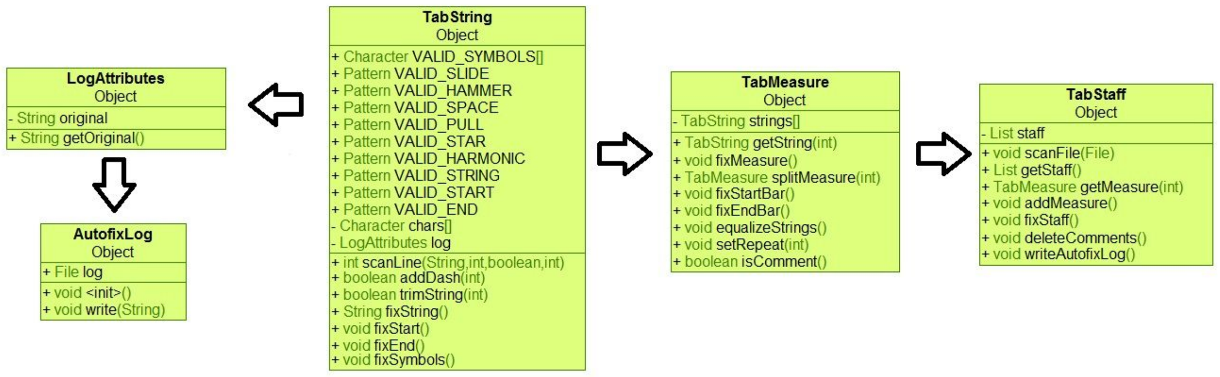


Figure 4: Class Diagram for Formatter.

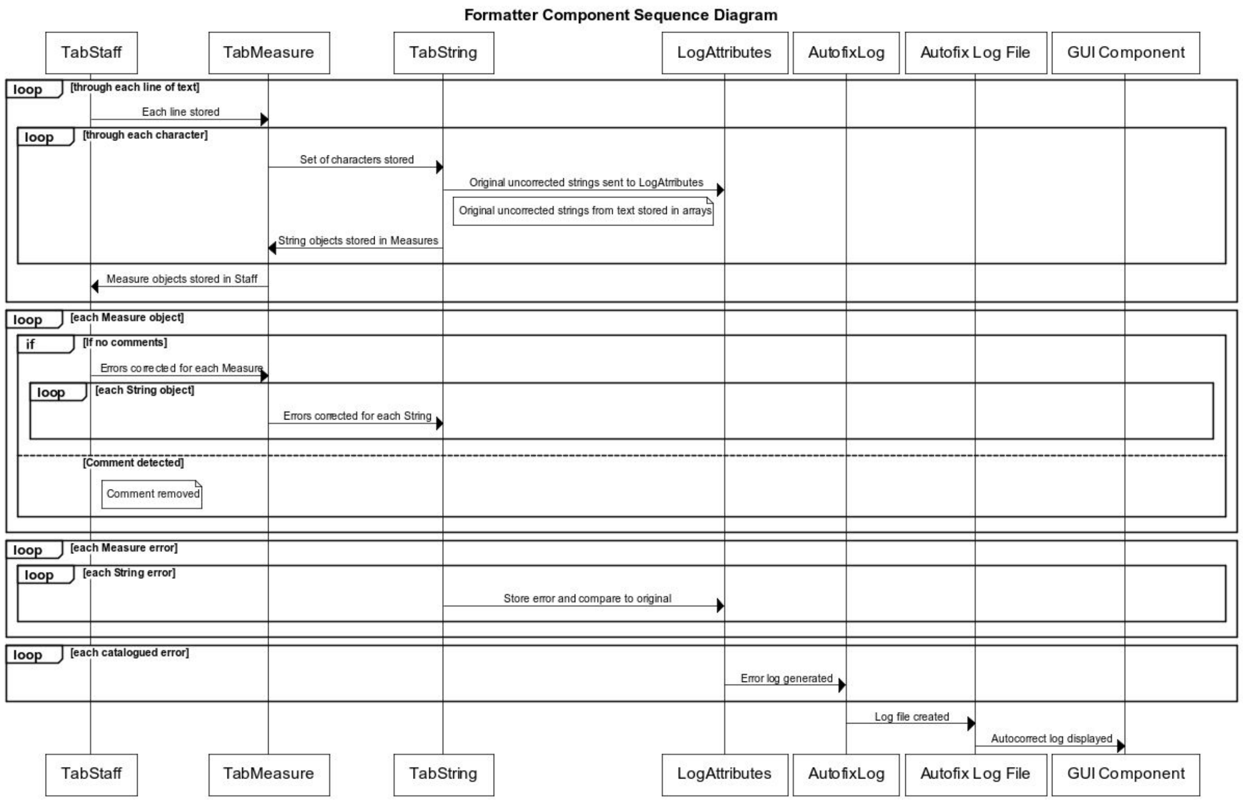

Formatter Component Sequence Diagram
TabStaff
TabMeasure
TabString
LogAttributes
AutofixLog
Autofix Log File
GUI Component
loop [through each line of text]
Each line stored
loop [through each character]
Set of characters stored
Original uncorrected strings sent to LogAttributes
Original uncorrected strings from text stored in arrays
String objects stored in Measures
Measure objects stored in Staff
loop [each Measure object]
if [If no comments]
Errors corrected for each Measure
loop [each String object]
Errors corrected for each String
[Comment detected]
Comment removed
loop [each Measure error]
loop [each String error]
Store error and compare to original
loop [each catalogued error]
Error log generated
Log file created
Autocorrect log displayed

Figure 5: Sequence diagram for Formatter.

4.2 Conversion

The TextToPDF uses drawClass, MusicNoteProcess, TabStaff objects to draw the music notes in the PDF document. At the beginning the music notes are passed in a list of strings that have previously gone through a series of processes such as fixing, deleting, adding, etc. to the individual music note by the TabStaff class so that its turned into a proper music note for the PDF. For the second step, music note strings will be converted to a list of MusicSymbols by using the MusicNoteProcess. Originally, previous versions of TextToPDF used to draw character by character by visiting a list of music note strings. However, this approach seems to be nonviable because it will involve a lot of case statements for each character. Consider this case ,suppose the we have a music note string:

?------------3---------|

It is unknown what character we have in ?. Possibilities include:

|------------3---------| case 1 : one vertical bar at the beginning.

||------------3---------| case 2 : two vertical bars at the beginning.

|||------------3---------| case 3 : three vertical bars at the beginning.

Since visiting is done character by character, visiting the first bar for all three case has three cases. The cases are: if the character after the bar is not a bar then one bar is drawn. The second assumption is if the second character is a bar, then we have to check for the third character for the third bar. If it is three bars, then three long bars in PDF have to be drawn. If it is not three bars, then we have to draw a thick bar to display two long bars. As seen from above, there are going to be a lot of cases for each character and has a formula of:

Number of cases = (character) * 4

This results in a lot of control statements. This will result in the class being too big, inefficient and thus making the program slow for the end user.

A different solution was needed. One of the solutions was to use finite state machine programming. The idea was to reduce the cases and have better efficiency than the previous method described. However, finite machine programming design definition states that it does not remember the last case it visited. This does not make it ideal for drawing music notes. Consider this example :

|------------------4------------|| ||-----------5s6---------|

If finite programming is used, then if double bars are entered for the first string, a thick line will be drawn. Next, if the second string (with 5s6) is on the same line, then it will enter a double bar state just like the previous string and draw a thick line. This will result in a large width thick line to be drawn, which is not considered to be a standard music note feature.

A third approach was constructed. Consider the following String:

|----------4s7------------------||

|--------35---------------------||

|--56---------------------------||

If a human was told to draw the music note on a sheet of music. They will see or take a ruler and measure the length of three bars at one time and draw a long horizontal line that has the same length as the three music bars. Next he/she would measure the length of one dash and count how many dashes are between the bar and the number and draw one long horizontal line, rather than drawing dash by dash. They would write the numbers and music symbols on the line and would do the same for rest of the music.

This last method was used because it is computationally faster and a more organized way of coding. Instead of considering the music measure/bar as a whole with one long case statement, only iterating to the first symbol, storing it and displaying it was done.

Similarly, the numbers will not print immediately because at small line spacing there might be a dash character after the number but the number font size is bigger than line spacing. So the horizontal line will pass through number. To avoid that the numbers must be printed after the music note. The method paintNumber() will print numbers after music note by storing each number, its xy coordinate using SymbolPoint class in a list and later paint them on music note. This method will be called inside DrawMusicNote.

The TextToPDF uses draw class to draw the lists from TabStaff. However, one problem arises is whether to draw music on the same line or on a new line. A solution would be to calculate the length of the music before actually printing the music note in order to determine if the length of that music note can fit in the same line or if it has to go to a different line.

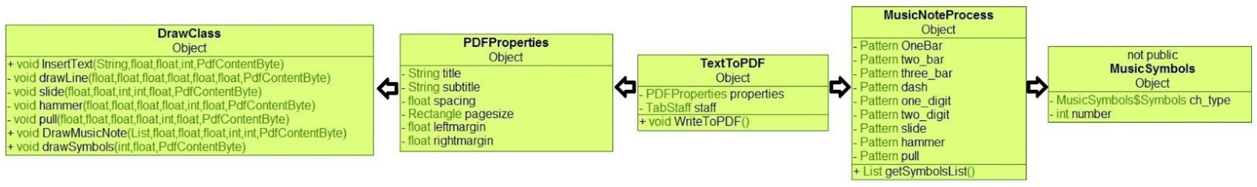

DrawClass
Object
+ void InsertText(String,float,float,int,PdfContentByte)
- void drawLine(float,float,float,float,float,float,PdfContentByte)
- void slide(float,float,int,int,float,PdfContentByte)
- void hammer(float,float,float,float,int,float,PdfContentByte)
- void pull(float,float,float,float,int,float,PdfContentByte)
+ void DrawMusicNote(List,float,float,float,int,int,PdfContentByte)
+ void drawSymbols(int,float,PdfContentByte)
PDFProperties
Object
- String title
- String subtitle
- float spacing
- Rectangle pagesize
- float leftmargin
- float rightmargin
TextToPDF
Object
- PDFProperties properties
- TabStaff staff
+ void WriteToPDF()
MusicNoteProcess
Object
- Pattern OneBar
- Pattern two_bar
- Pattern three_bar
- Pattern dash
- Pattern one_digit
- Pattern two_digit
- Pattern slide
- Pattern hammer
- Pattern pull
+ List getSymbolsList()
not public
MusicSymbols
Object
- MusicSymbols$Symbols ch_type
- int number

Figure 6: Class diagram for Conversion. Arrows indicate calls to a class.

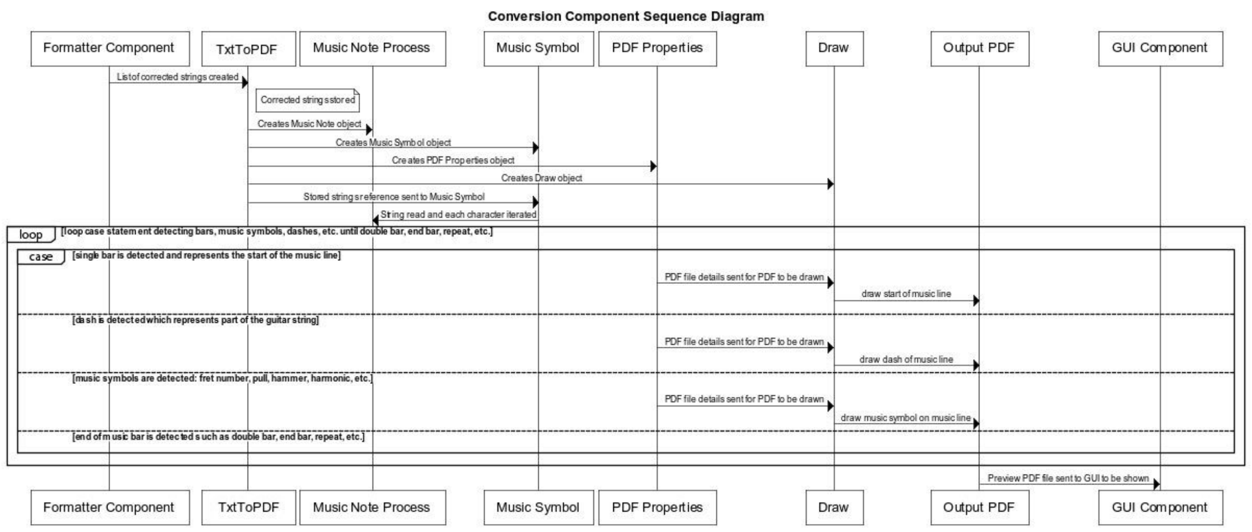
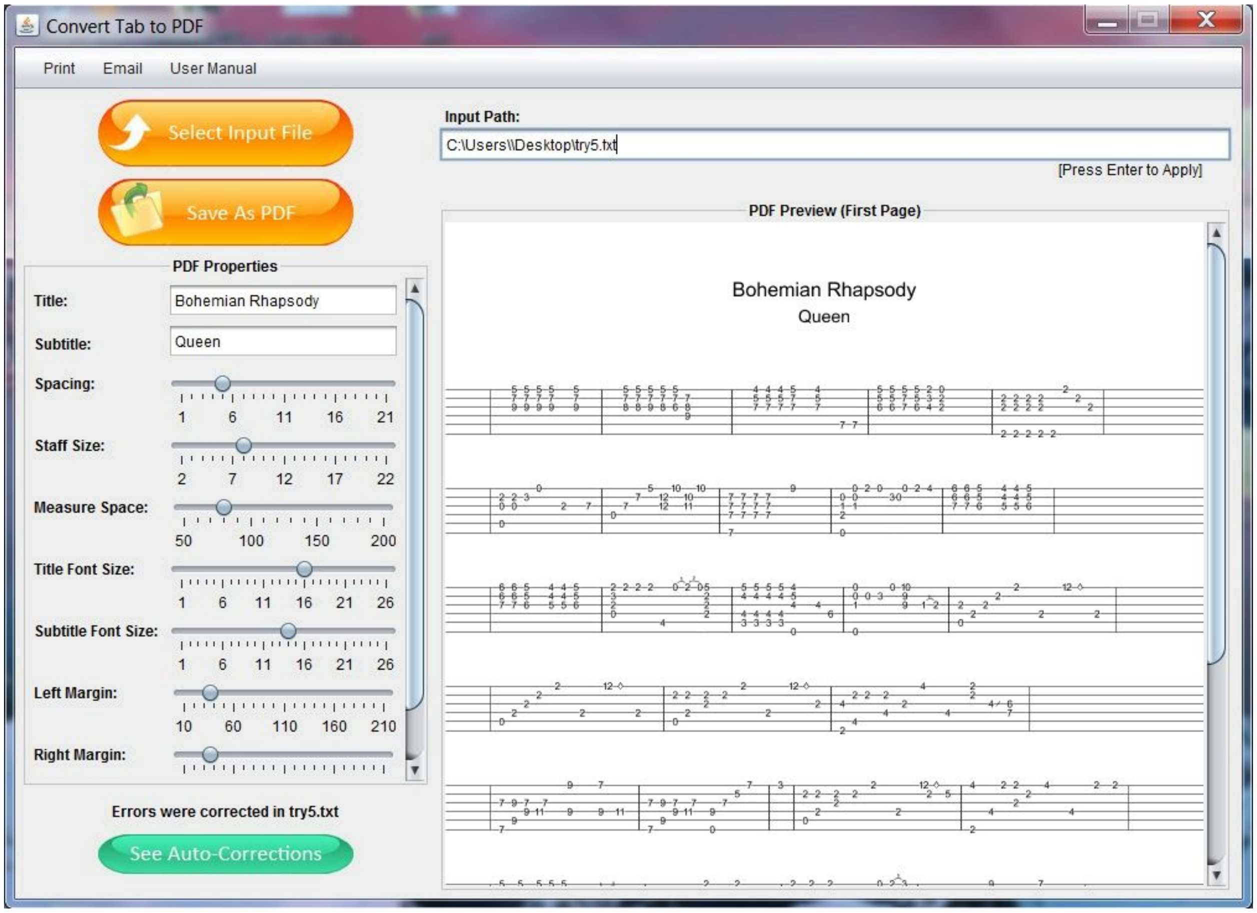

Conversion Component Sequence Diagram
Formatter Component
TxtToPDF
Music Note Process
Music Symbol
PDF Properties
Draw
Output PDF
GUI Component
List of corrected strings created
Corrected strings stored
Creates Music Note object
Creates Music Symbol object
Creates PDF Properties object
Creates Draw object
Stored strings reference sent to Music Symbol
String read and each character iterated
loop
[loop case statement detecting bars, music symbols, dashes, etc. until double bar, end bar, repeat, etc.]
case
[single bar is detected and represents the start of the music line]
PDF file details sent for PDF to be drawn
draw start of music line
[dash is detected which represents part of the guitar string]
PDF file details sent for PDF to be drawn
draw dash of music line
[music symbols are detected: fret number, pull, hammer, harmonic, etc.]
PDF file details sent for PDF to be drawn
draw music symbol on music line
[end of music bar is detected such as double bar, end bar, repeat, etc.]
Preview PDF file sent to GUI to be shown
Formatter Component
TxtToPDF
Music Note Process
Music Symbol
PDF Properties
Draw
Output PDF
GUI Component

Figure 7: Sequence diagram for Conversion.

4.3 GUI



Input Path: C:\Users\\Desktop\try5.txt

Title: Bohemian Rhapsody

Subtitle: Queen

Spacing:

Staff Size:

Measure Space:

Title Font Size:

Subtitle Font Size:

Left Margin:

Right Margin:

Errors were corrected in try5.txt

Bohemian Rhapsody

Queen

The basic layout for the main graphical user interface was chosen because it allows the user to distinguish between each panel, as each panel itself contains it's own set of information. Upon running the program, the only editable fields are the select button and path to the file text field, as the user flows through the experience more options become available, this is all explained below.

The upper most left panel holds 2 buttons: Select input file, and save as. The Select input file button displays a file selector where the user may select the file they would like to convert. Upon selection the file path will be displayed in the panel directly to the right of the button, then the conversion code will run. The save button, editing panel, and preview panel (the designs of the latter 2 are explained below) only become visible when the conversion code has been successful. These buttons provide an intuitive experience for the user to enjoy the main functionality of the application without having to edit the pdf.

The panel located directly to the right of this, which takes up the entire right side, is the preview panel. This panel contains a text field holding the location of the selected text file (updated after selecting the browse button, or user may type the file path). Also this panel holds a preview (image) of the pdf after the user selects the convert button. If our application detects any errors that we could not fix automatically in place of the preview an error message is displayed at which point the editing panel and save as button stay blank.

The next panel located just below the buttons panel holds all the information to edit the pdf, this panel only becomes visible when the user selects the convert button. It includes a text field for changing both the title and subtitle which updates the preview upon pressing the return (Enter) key. As well, sliders to change the line spacing and font size of the tab characters on the page which also update the preview when the slider is released. Updating the pdf after every detail has been changed allows the user to see chances occur in real time, and flow through the editing experience with ease.

The final panel is the autocorrections panel, this panel only becomes active if our program has made any chances to the input which reflect in the output pdf. The panel has only one button which the user may select to view in a new window what the necessary chances our program has made to their input.

When auto corrections were made, the user may select the "see auto-corrections" button at which point a new window, auto correction log, will be opened that displays the corrections that were made in order to convert the tabs into pdf format. This is helpful for the user to see if what they've entered is valid.

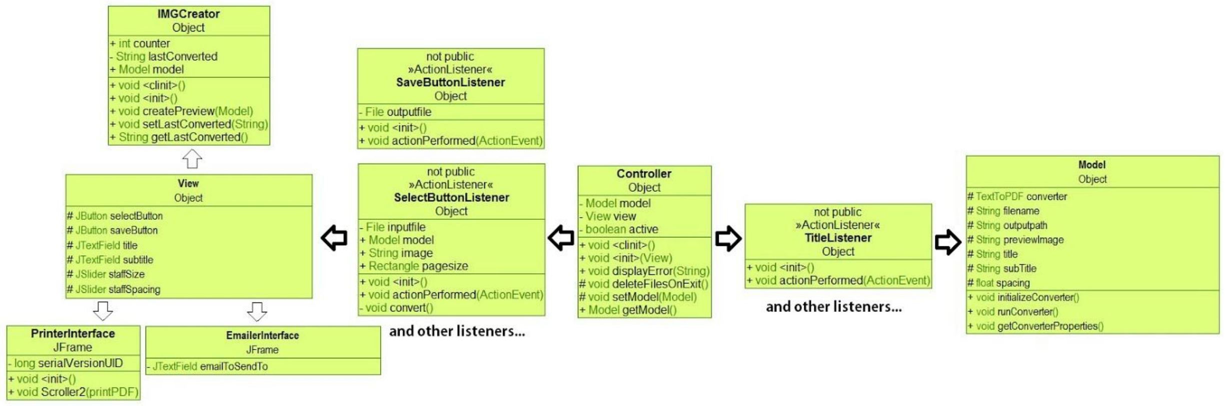


Figure 8: Class diagram for GUI.

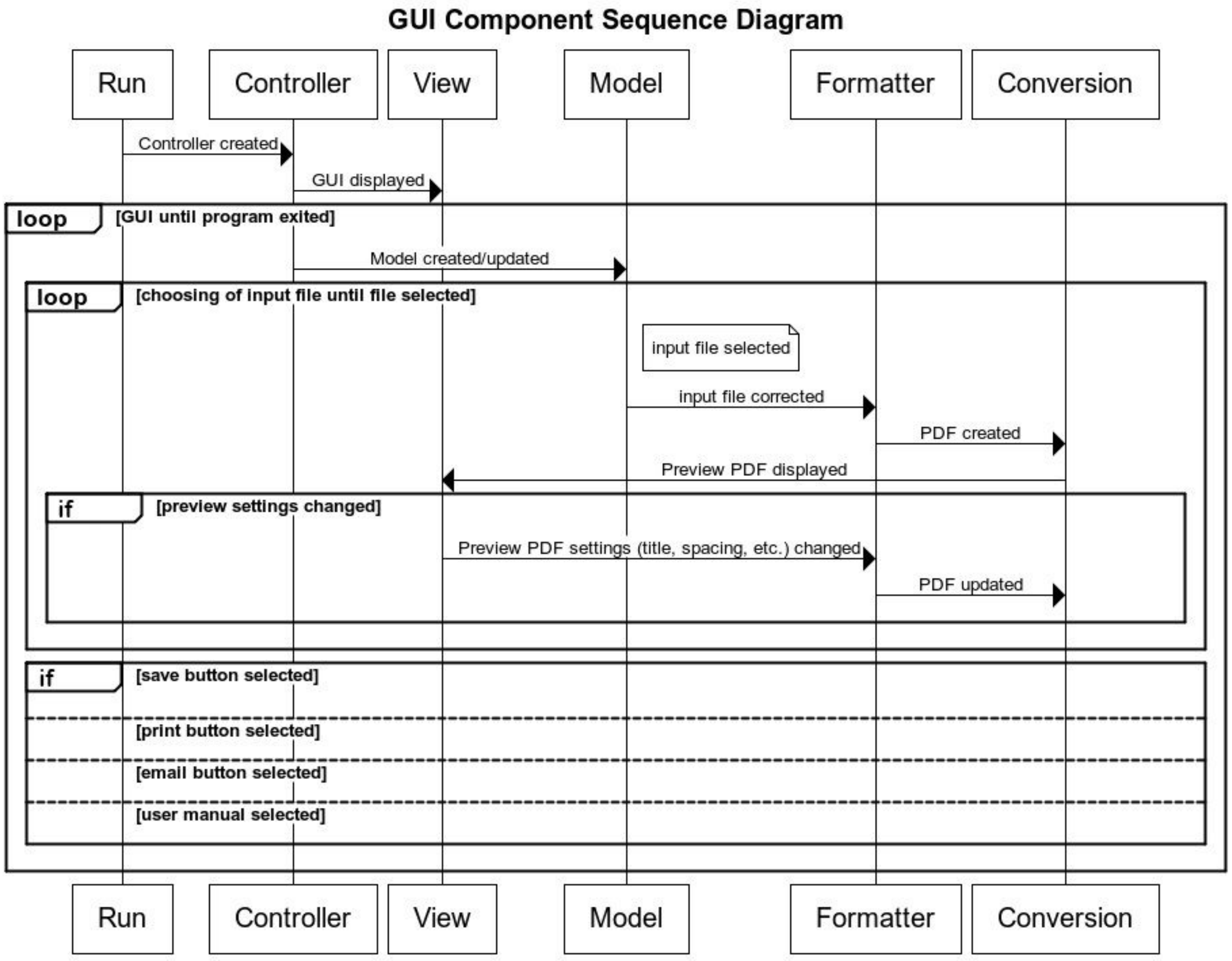


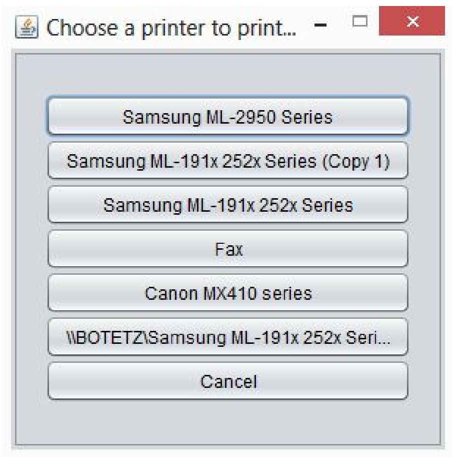
Figure 9: Sequence diagram for GUI.

4.4 Additional Features

Emailer
The emailer is implemented in the View class of the GUI. The View class creates a new instance of the EmailerInterface class when the "Email" button is clicked within the menu bar. The EmailerInterface class creates a new JFrame with basic Java swing components such as a JPanel, JLabels, and JButtons. The cancel button disposes the newly created JFrame and the login button would create a new instance of the getSentMessages class when clicked. The getSentMessages class takes in two parameters: a username of String type, and a password of String type. The getSentMessages class then attempts to make a connection to the user's Gmail account by using various functions found in the Javax Mail library. A connection is made by using "imap.gmail.com" as a host to make a connection. If a successful connection is made then the class will retrieve all the email addresses the user has sent mail to so that it can later create an address book. If an unsuccessful connection is made then a Boolean false is returned to the EmailerInterface class which would then display an invalid message to the user indicating that his/her email/password was typed in wrong or there was no internet connection. Otherwise, the EmailerInterface class would then move on to its next screen where the user can type in the name of the email address he/she would like to send mail to, a subject line, message, and attachment (ideally the converted PDF file). The EmailerInterface switches screens by removing all Java swing components from the JPanel and adding in new ones. Additionally, there is a "Contacts" button that when clicked will create another JFrame that displays all the user's contacts by using the returned data from the getSentMessages class. The "Send" button in this panel creates a new instance of the Emailer class and calls the sendEmail method when clicked. The Emailer class takes in two parameters: a username as a String type, and a password as a String type. Furthermore, the sendEmail function takes in five String parameters: a path of what file to send, the new name of the file attachment, the email subject line, the email message body, and the email address to send to. This method sends the email by connecting to the "smtp.gmail.com" host, the "587" port, and using various methods from the Javax Mail library.

Printer
The printer is implemented in the View class of the GUI. The View class creates a new instance of the PrinterInterface class and the printPDF class when the "Print" button is clicked within the menu bar. The PrinterInterface class creates a new JFrame with a JPanel and JButtons for every printer in the network. The printPDF class takes in one String parameter which is the pathname of the PDF file that needs to be printed. The printPDF class creates a PrintService using the Javax Print library so that it can connect to the network and see what printers are available. When a printer is selected the PrinterInterface class would call the printFile method located in the printPDF class. The printFile method sends the print job by creating a DocPrintJob object

and calling its print method.


Choose a printer to print...
Samsung ML-2950 Series
Samsung ML-191x 252x Series (Copy 1)
Samsung ML-191x 252x Series
Fax
Canon MX410 series
\\BOTETZ\Samsung ML-191x 252x Seri...
Cancel

User Manual
The user manual is implemented in the View class of the GUI. The View class creates a new instance of the userManualInterface class when the "User Manual" button is pressed. The userManualInterface class creates a new JFrame with two JPanels. One JPanel contains the JTree that allows users to navigate through different sections. The second JPanel contains numerous JLabels that are used to explain each section of the user manual. The pictures found in the user manual are JLabels that call the setIcon method so that it will have an image set to it. The setIcon method takes in an ImageIcon parameter, and an ImageIcon is just a child that extends off of the BufferedImage class. Additionally, each node in the Java tree contains an action listener that scrolls down to the location of that step. This technique of scrolling is done by calling the setViewPort method in the JScrollPane component that was added to the panel.

Java Swing EDT

The program runs Java Swing, which runs mostly in the Event Dispatch Thread (EDT). The EDT is a special thread, which was created to process the events of the user interface in a queue process. Each event is queued to the EDT to be handled and processed; thus, events occur one by one ensuring the safety of the swing components. The EDT handles the repainting and listeners of the swing components in the program. The component calls the EDT for a change or for a particular event, the event is queued, and the appropriate actions are taken when the queue has arrived at said call. While swing components run on the EDT unpredictable and irreproducible errors will not be encountered.

The graphical user interface of the program is first initialized by using the EDT, specifically by invoking the SwingUtilities.invokeLater() method. The method runs a runnable, which initializes the graphical user interface and awaits user command. This ensures that all interface processes are handled by the EDT so that component listeners and paint calls are performed properly.

Multithreading/Concurrency

This program is a concurrent software. While the program converts the given text file which contains tablature, the user interface will continue to respond to user interaction. This ensures that the user can move the program window, select another file, or even exit while the program is converting. Furthermore, while the program is converting a new input file, a loading bar is shown to communicate to the user that the program has received the user's input and is processing it at the current time. When changes are made to an input file through the page properties panel, the cursor is set to a loading cursor when pointed at the preview panel. However, the user is still able to open the auto-fix log if they wish to do so while the change in the preview occurs. This helps the user to know that the program is working and is responding; essentially taking away the frustration of working with a seemingly non-responding program.

All of this is made possible by using more than one thread. While the swing components work from the aforementioned EDT, a background thread is invoked to handle the converting process apart from the EDT. The background thread is created using the help of javax.swing.SwingWorker's methods: doInBackground(), process(), and done(). When the user chooses a file, the doInBackground() method is used to create a background thread that converts the text file into a pdf and the pdf into an image. Before the conversion begins; however, data is sent to the process() method to let it know about the progress of the conversion. That data is translated into the indeterminate loading bar and loading cursor as the image is being updated. Finally, when the conversion is finished, the done() method is used to update the view about the changes which include the page properties of the file, the preview of the document, restoring of the cursor, and the hiding of the loading bar. Through multithreading, the user interface communicates to the user of the ongoing processes and does not freeze during the conversion to ensure the user that the program is still responding.

IMGCreator Class

The IMGCreator class creates an image of the first page of the converted pdf to be previewed to the user to see how the changes look. The class is dependent upon the jpedal library to create a .png image of a pdf.

The class accesses the input pdf file path from the model class and converts the first page into a BufferedImage. The BufferedImage is afterwards rescaled using the AffineTransform class with the same transparency. The final image is then saved temporarily for the user to view on the preview panel to decide whether or not to change the output document.

5. Maintenance Scenarios

Maintenance scenarios include updates to the software that fix bugs and add features. These updates will be delivered through a website where the user installs a new release of TabToPDF after uninstalling a previous version. Previous versions will remain available in case the user preferred older functionality. A valid licence code that was purchased would be needed to access new version of the software. Email updates of when a new version is available will be sent.

In order to increase the user/client base of TabToPDF and make it more successful, the first change will be to add more instruments. Adding functionality for all types of guitars and then branching out to other classes of instruments until the software can make sheet music for any instrument is a priority. An example would be to first implement support for 4 string guitars or ukeleles and then developing Conversion for percussion until support has been achieved for bands or orchestras.

Secondary features that would be implemented would be to increase the streamlining and multimedia functions. For example, allowing more functionality for emailing or sharing music on social media sites. A more ambitious feature would be for the sheet music to be played using MIDI software or sound so that composers can create music.

Implementing these changes would not be difficult since the software has been designed to be very modular. To implement additional guitars would require minor changes to Conversion code, or even only additions and Formatter code already has features supporting such a design due to a constant for how many music bar lines there should b. For example the user would specify what type of guitar, or instrument, the input text file has been formatted for. Ignoring a string would not be difficult in the case of 4 string guitars since the code would only need to skip a line of text in the input file (most likely the 5th-6th string if the user wishes to use their input text file immediately to create 4 string output PDF). Otherwise, the input text file formatting guidelines can be developed for the user to create proper text files that will be autocorrected and converted to PDF to support 4 string guitars.