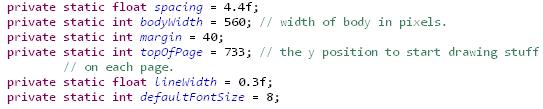
**Design Document for TAB2PDF**

This document pertains to the inner workings of the Team Solo TAB2PDF program and how each and every class in the program along with their respective methods work together to achieve a smoothly operating program that converts an ASCII file in .txt to a PDF tablature that the user can then edit from the GUI and use as his music sheet.

There are 6 classes that pertain to the base system called “Creator.java, InputUtil.java, PDFCreator.java, PreviewCreator.java, TAB2PDF.java, and Tablature.java”. Creator.java is the program starter where everything that is required to run is initialized in.

The first thing to mention would be the stave storage string where it is split up between all the small staves in the ASCII document and then is converted into one big stave as a string but before this, a static stave buffer is used to ‘ready’ a stave or bar in order to shorten run time.

There are different variables for a bunch of different methods such as ‘spacing’, ‘bodyWidth’, ‘topOfPage’, ‘lineWidth’ and ‘defaultFontSize’ each with their own initialized values:



Spacing is set for now only to compile the program but it is changeable.

* ‘bodyWidth’ is the width of an A4 piece of paper in PDF format.
* ‘topOfPage’ is how far down the page should the first stave be drawn in pixels
* ‘lineWidth’ is the width of each line drawn in the PDF
* ‘margin’ is how far into the page from the left does the first filled in stave need to be placed in pixels
  +  this is an example of the margin (the space from the left to the first bar) and the ‘topOfPage’ is the blank space at the top

Here are the methods that were created in the Creator.java class:

* getSpacing() which returns the current spacing being used
* getDefaultFontSize() which returns the current font size for printing the notes
* setSpacing(spacing) which is used to set the current spacing to ‘spacing’ but only if spacing is less than 80 pixels otherwise it stays as 80
* setDefaultFontSize(fontSize) sets the font size to be used in the PDF with the original being 8
* createTab(doc, contentbyte, body) which is the called method to produce a new PDF tablature and this method calls on all the methods necessary to create a new PDF. This method is only called when creating a brand new PDF, not when updating an existing one. It requires a Document, PDFContentByte and a String to be called properly
* drawTablature() is responsible for drawing the actual PDF. It is called the first time the PDF is created and any other subsequent time after that. It assumes that the staves have already been concatenated into one large stave to be partitioned into smaller staves which will fit within the width of the page
* drawTripleThinBars(xPos, yPos) draws three vertical bars where xPos is the position of the first bar relative to the last character in the token multiplied by the line spacing
* curveAndLetter(token, yTune, xPos, yPos, fontWidth, letter)
  + token is the token to be parsed
  + yTune is used to tune the vertical position relative to the original
  + xPos is the horizontal position of the symbol to be drawn
  + yPos is the original vertical position of the symbol to be drawn
  + fontWidth the width of the character calculated which will be used to decide where the bar lines start after the character and where they end before the character
  + letter is the actual string being used here
* drawPullOff(token, yTune, xPos, yPos, fontWidth, letter) is basically the same as curveAnd Letter()
* calcPosRelToEnd(posFromEnd, xPos) finds the position relative to the last character of a token
* drawLeftMargin(xPos, yPos) draws the left margin of the document assuming that the xPos is equal to the margin width
* drawLagatoSlide(token, yTune, xPosTok, yPos, fontWidth) which draws the legato slide which is a digit followed by a slash through the bar-line followed by another digit
  + each individual component is the same as for curveAndLetter()
* drawSlash(xPosTok, yPos) is a helper method for the drawLagatoSlide which draws the slash going through the bar-line
* drawDiamond(xPosTok, yPos, diamondSpace, diamondWidth) draws the diamond shape for the volume swells.
* singleDigitLineSurround(xPOsTok, yPos, fontWidth) is used to draw a line up to the character and starts right after the character width has ‘ended’
* drawText(yTune, xPos, yPos, text, fontSize) draws the text to the specified position with components similar to that of curveAndLetter()
* drawCircles(xPOs, y, offset) draw dots at the specified x and y coordinates
* drawBeginRepeatSymbol(x, y, side) draws a thick circular dot at the given x and y position
* drawThickVertBar(x, y) draws a thick vertical bar at the specified x and y coordinates
* drawThinVertBar(x, y) draws a thin vertical bar at the specified coordinates

All of the above were to do with the general special characters and the fine-tuning of the PDF drawn tablature the rest from here on is related to arranging the staves

* getBigStave() returns the big stave as a string
* makeBigStave(body) makes one big stave with all the measurements on it
* separateStaves(s) separates all the staves in s into separate strings
* removeConsecutiveBars(staves) takes an array of staves and removes consecutive vertical bars so that the big stave is a valid musical stave e.g:
  + | - - - - - | - - - - - | - - - - - | - - - - - |

Instead of

* + | - - - - - | - - - - - || - - - - - | - - - - - |

which occurs after concatenating staves together

* getOrganizedStaves() which returns a list of staves partitioned in a way such that they will fit on the page
* concatStaves(staves) concatenates all the staves into one long stave which is returned as an array of strings where each string is a separate line of the stave
* reorganizeStaves() takes the big stave and partitions it into 1 or more staves such each stave contains the maximum number of measures while each stave remains with the boundary of the body width assuming the big stave has been initialized and the first measure has no more than 3 vertical bars at the start
* makeLastStave(lastIndex, i) checks to see if the end of the big stave has been reached
* indexOfLastMeasure(lastBarLastMeas, firstBarCurMeas) returns the index of the previous measure. If no previous measure is found or the current measure too large to fit within the bounds then -1 is returned. This method specifically returns the index of the last vertical bar at the end of the preceding measure
* isInBounds(lastIndex, curIndex) checks to see if the space between the two indices is within the bounds of the page
* cutStave (lastIndex, curIndex) copies a partition of the big stave into a stave buffer to be copied into the final list of organized staves assuming that bigStave has been initialized, lastIndex must be less than curIndex and both bust be natural numbers
* getRepeatNum(line1, line2) returns the repeat number embedded in double vertical bars if no number is embedded returns -1. Example:
  + |2

||

||

||

Will return 2 while

* + ||

||

||

||

Will return -1

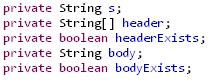
* countEndBars(line) counts the number of consecutive vertical bars counting backwards from the end of the last line assuming all vertical bars are located within the last 5 indices of the line and all measures have a length greater than 5
* countBeginBars(line) counts the number of consecutive vertical bars counting from the start of the first line with the same assumption as countEndBars() but instead of last 5 indices it’s the first 5
* buildStave(lines) takes an array of lines and builds a stave from it
* splitLines(stave) splits a stave into separate lines
* getLine returns the i’th line of the given string where 0 is the first line. If there is no i’th line then ‘null’ is returned
* countLines(s) counts the number of line in the string
* drawHeader(canvas, header) draws a stylish header for the created document

This concludes the method in the Creator.java method with a brief explanation of each method. The next class on the list would be the PDFCreator.java which contains the writePDF(filename, tab) method which takes the filename to be used in writing the PDF and the Tablature tab to be created. This method uses the drawHeader() method from Creator.java and the createTab() method from there as well.

There is also a utility class that was created which is used to read in the ASCII files called InputUtil.java and converts into one big string which it then returns using the method openFile.

Another important class used is the Tablature.java class which is the document being built to be writing as a PDF file. It contains the original ASCII as a single string then splits it into individual parts from the file. For example it would hold the header as separate data from the music tabs.

This class contains the following methods and variables:



These are the initialized variables used to create the tablature and are used in the methods below.

* The constructor Tablature(String s) initializes the variables header and body and sets the ascii to be read as ‘s’
* setAscii(s) returns true after setting ‘s’ as the ascii to be read
* getAscii() returns ‘s’
* getHeader() returns ‘header’

There are also test used to tell if the inputted ASCII file has a header which includes a title, subtitle and spacing

* checkHeaderExists() returns true if all of the above are present
* parseHeader() parses the title, subtitle and spacing of the ASCII file
* hasHeader() returns the variable headerExists
* hasBody() returns the variable bodyExists
* parseBody() parses the rest of the ASCII file
* getBody() returns ‘body’
* toString() returns ‘s’

The next class is an extra features class that is placed in the base code as it needs to create it before using it which is the PreviewCreator.java class which creates the preview first before drawing it anywhere and this class basically uses a ByteArrayOutputStream which returns a ByteArray image of the created Tablature

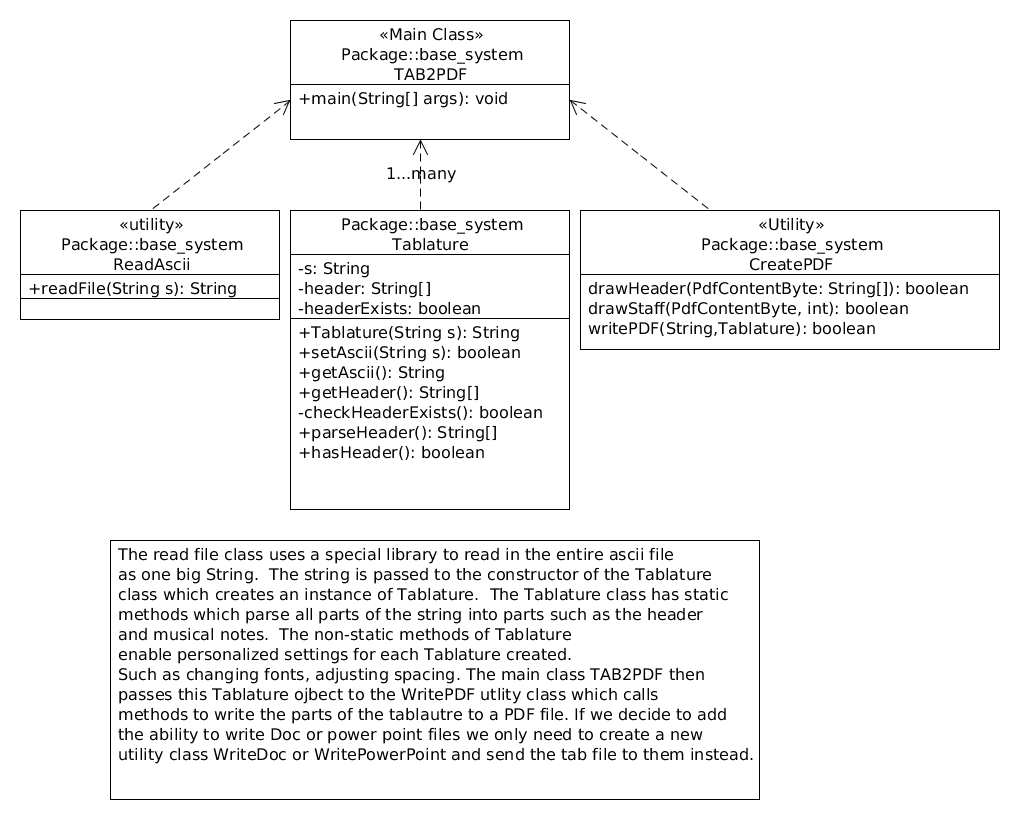
The final class in the body of the code is the main class called TAB2PDF.java which is the main class of the program that runs the base system which is what is called on in the GUI to start the conversion.

There is another section of classes called the extra features package which, currently, contains the PDFPreview.java class which basically returns an image of the Preview and is the method called upon by the GUI to show and image of the to-be-created PDF. This class calls on the aforementioned class called PreviewCreator.java so they work in unison.

This package will soon contain another class that will be called PDFPrint.java which prints the created PDF directly from the program to save time for the user.

Another package that was created in order to make this program fully functional is the GUI package which contains the GUI.java class and this is the user-customized GUI and it enables the user to alter many things from the file he wants all the way to the spacing in the document that is used.

Here is a UML diagram of the currently used classes and their respective methods that work together to make sure the conversion is done smoothly and quickly:



Also available is a quick diagram of how the program goes through the steps from the .txt file to a .pdf file:

