# System Design Document

## TAB2MXL

Group 1: Lian Attily, Alborz Gharabaghi, Robert Mealey, Derui Liu, Isaiah Linares

| | |
|---|---|
| Document ID | SYSTEM DESIGN-v0.2 |
| Version Number | 2.0 |
| Issue Date | April 13th, 2021 |
| Classification | Public |

# Table of Contents

# 1 Introduction

Our system is able to convert tablature into a MusicXML (.xml) format. This format is able to be opened in any sheet music display application.

## *1.1 Purpose of Document*

The purpose of this document is to describe the design of the TAB2MXL (T2X) Software as described in the T2X Software Requirements Document. This document is intended to be read and understood by developers that would like to understand how T2X works, as well as developers that intend on contributing code to the software.

## *1.2 Document Scope*

This document describes the design details of the T2X Software. This includes UML Use Case Diagrams, Sequence Diagrams, Class Diagrams, Object Diagrams and an Activity Diagram. This document will not explain how the design was implemented in Java.

## *1.3 Methodology, Tools, and Approach*

- Coded in Java (**JavaSE 12**)
- Automatically built and tested with **Gradle** . Dependencies include:
    - Junit
    - TestFX
    - JAXB
    - JavaFX (OpenJFX)
- GUI is coded using the **JavaFX** 15 framework and designed with **SceneBuilder** (styled with CSS)

# 2 Detailed Design

## *2.1 Use Case Diagram*

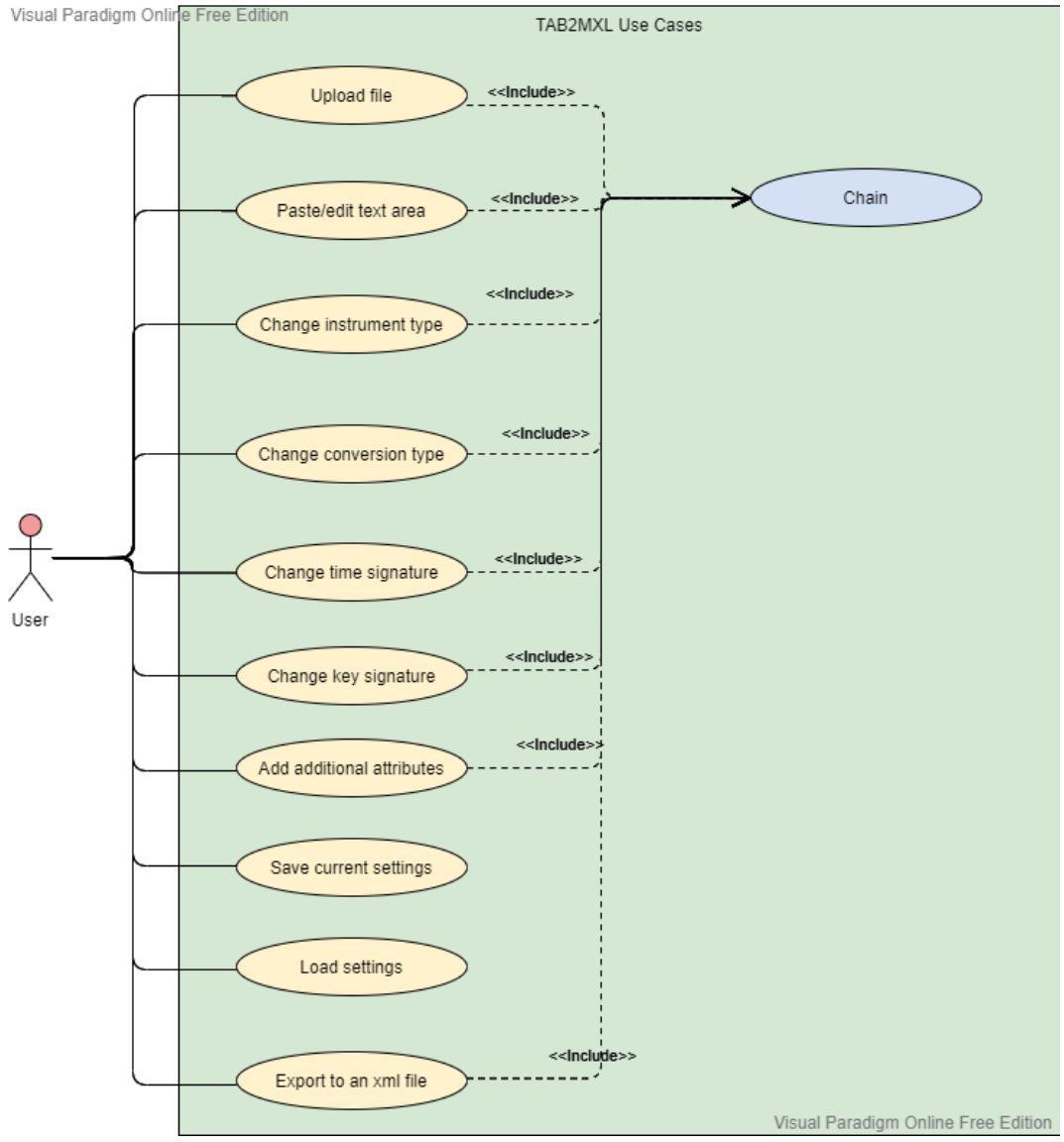
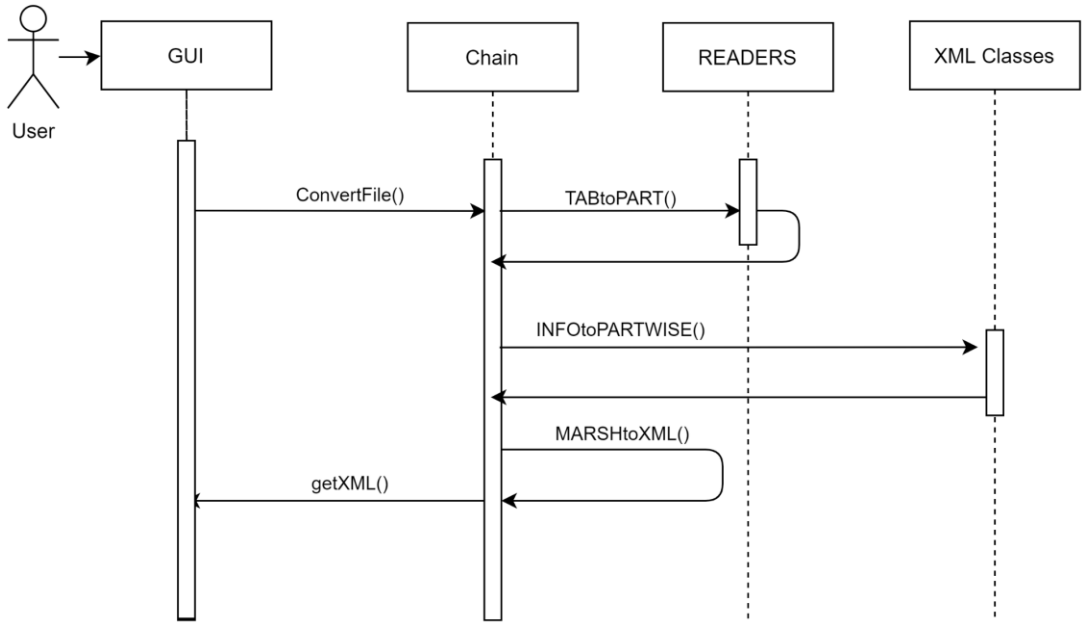
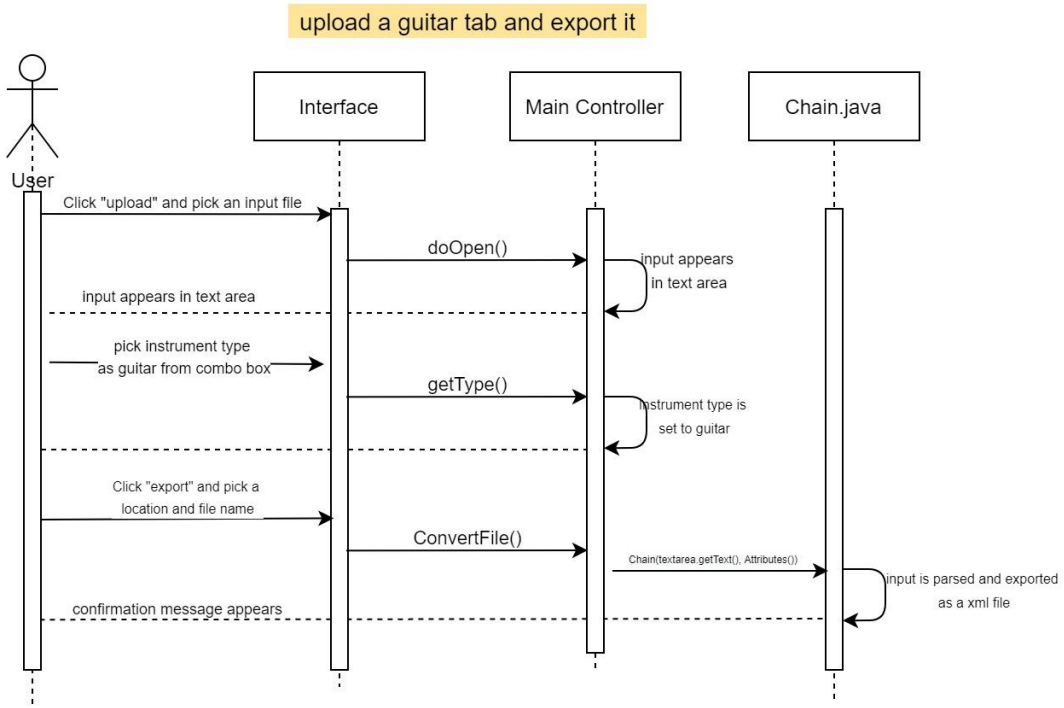



*Figure 1* describes the main features and functionalities of the software. The *Chain* class acts as a medium connecting between the GUI (Graphic User Interface) controller and the numerous classes involved in the parsing and conversion of the tab to an xml file.

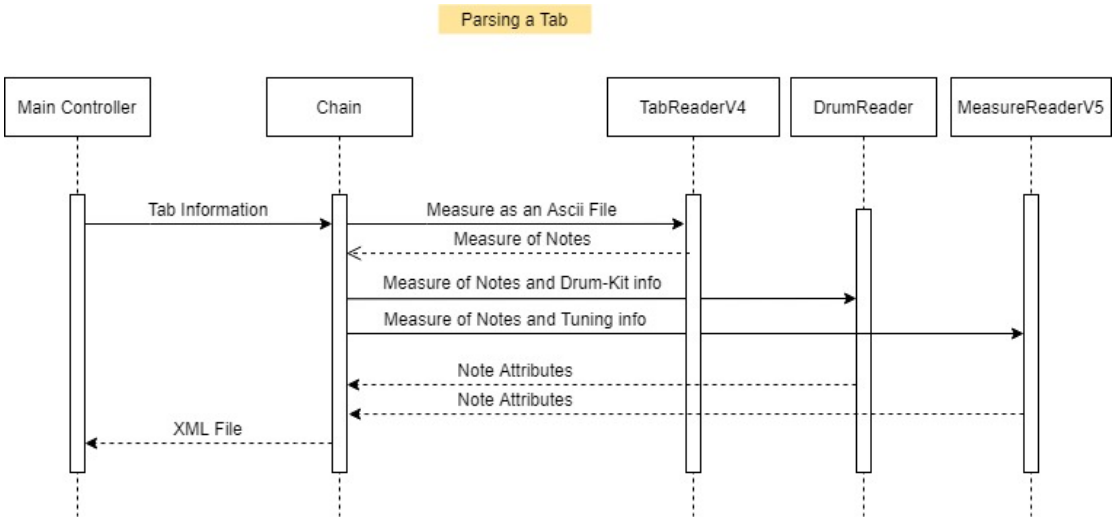
## *2.2* UML Sequence Diagrams



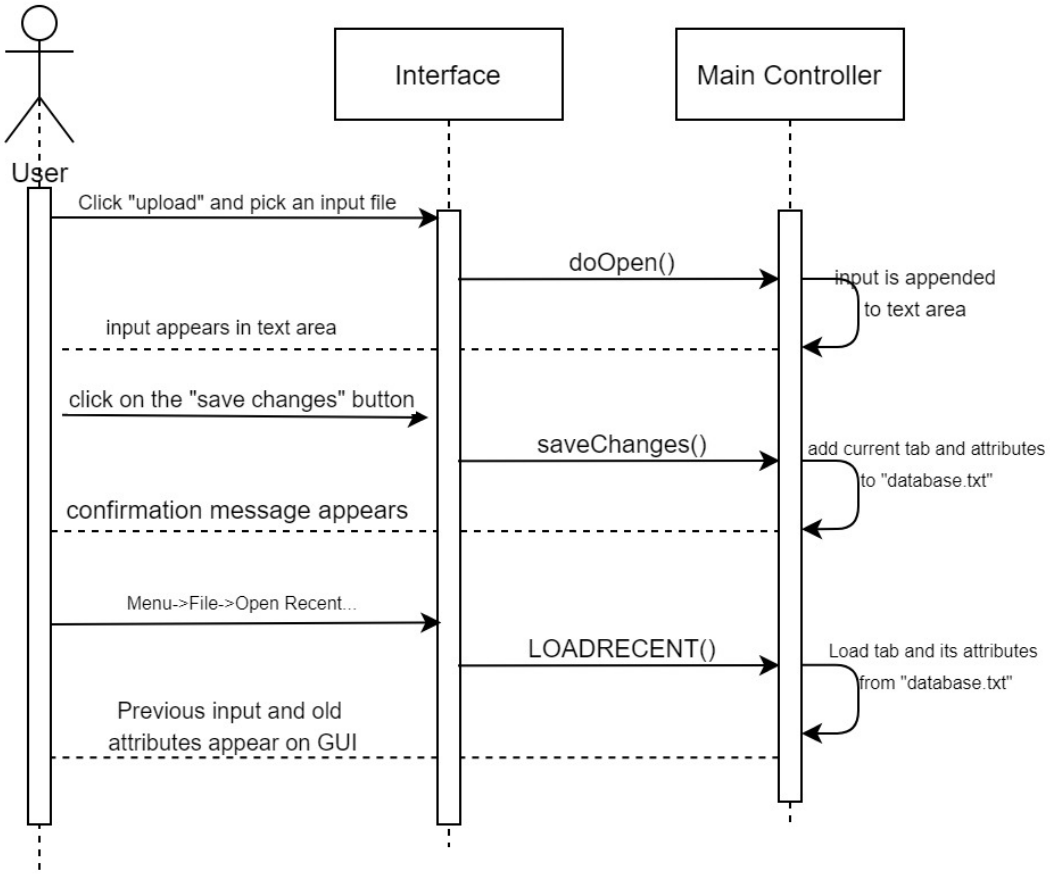

The sequence diagram above, demonstrates the zoomed out view of Chain. Chain acts as a timeline to run though all the aspects of parsing a given tab to exporting a string that contains the xml formatting. Chain receives the TAB from the GUI, sends it to the Drum and Guitar Readers respectively, converts the data into XML objects and then returns it to the GUI.




This sequence diagram illustrates the fundamental use of the T2X software. The user provides the software with an input tab - in this case the method doOpen() opens a file chooser that allows the user to pick a *.txt file from their machine. Then, the chosen file will appear in the editable text area on the GUI. The user may then pick an instrument type (i.e, guitar) from a combo box if the **automatic instrument detector does not work**. This invokes the getType() method which sets the instrument type to “Guitar”. By clicking the export button, the convertFile() method is invoked, prompting a save dialog box. ConvertFile() creates a Chain object - a mediator between the GUI controller and the backend parsing that creates an *.xml file at the specified path on the PC. A confirmation dialog pops up to inform the user that the conversion was successful.

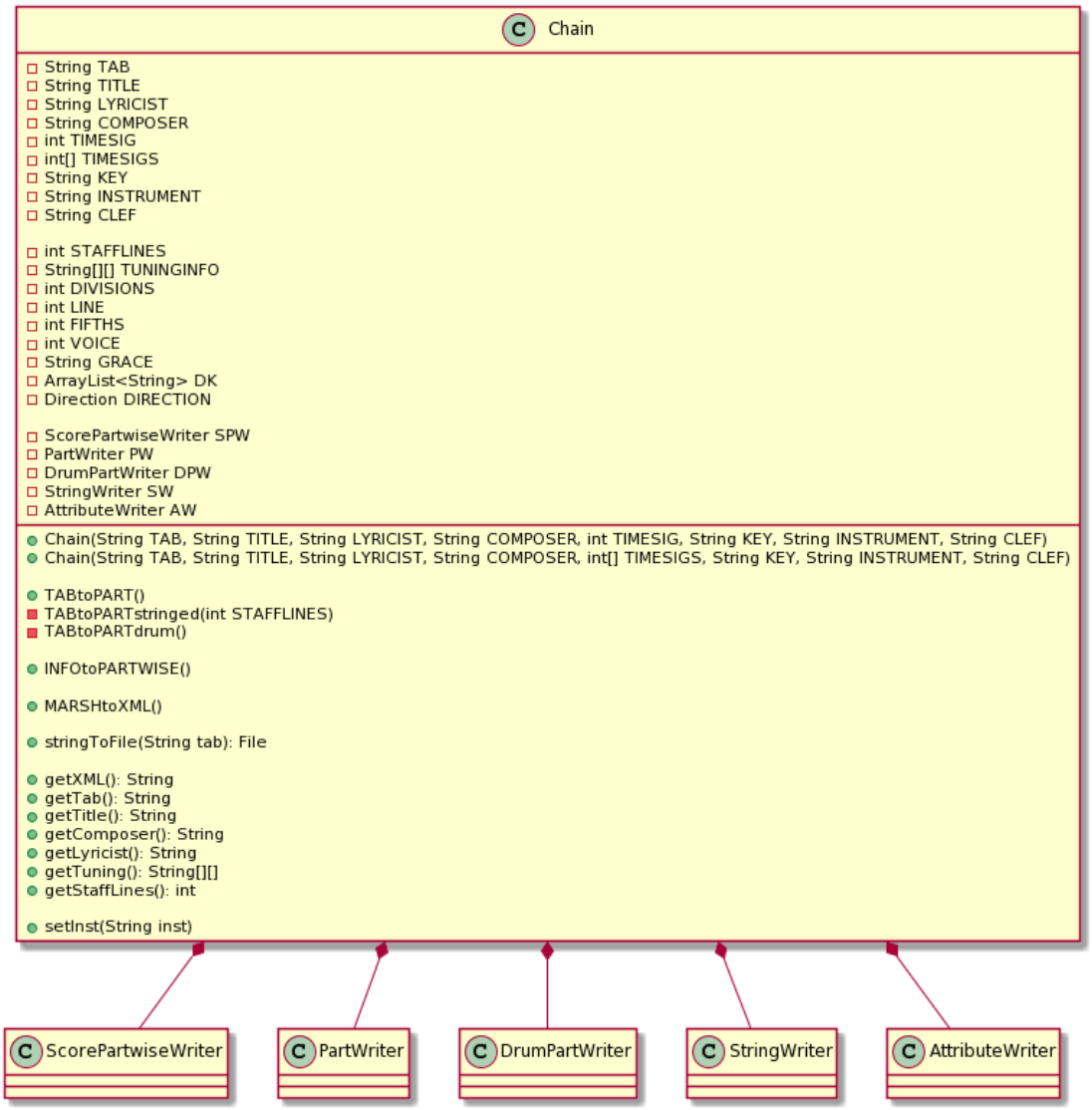



This diagram shows the interaction between different classes in the application when tablature in an Ascii file is parsed into a MusicXML file. It shows every step in turning a piece of tab from an ascii file into a music xml file. First all tab information: File path, Piece Information (lyricist, compost, etc…), Time signature, and Instrument, is used to create an instance of Chain. Chain can then use the TabReaderV4 class to get String arrays of each Measure. For each  Drum or Guitar/Bass measure, the Measure is sent to the DrumRead or MeasureReadV5 respectively. These two readers then give all the attributes of each note in that array. Chain will store this information using Object for each element in a MusicXML file. Once all the information is parsed, Chain will marshall it into MusicXML notation and return it to the MainController.

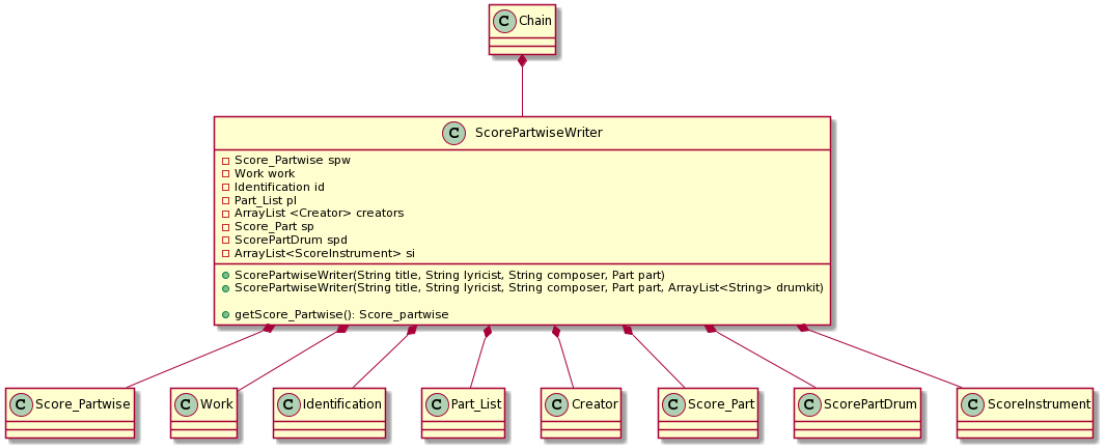



The diagram above illustrates the general flow of saving and reloading a file and its attributes (time signature, key signature, conversion type..etc). The user provides the software with an input tab - in this case the method doOpen() opens a file chooser that allows the user to pick a *.txt file from their PC. The chosen file will appear in the editable text area on the GUI. The user will then click on the “save changes”" button, invoking the saveChanges() method in the GUI controller class, which saves the tab and all its attributes to a fake database. To reload the file and its attributes, the user should click on Menu -> File -> Open Recent. This invokes the LOADRECENT() method that loads and sets the textarea and its attributes from the fake database, which appear to the user in the GUI.

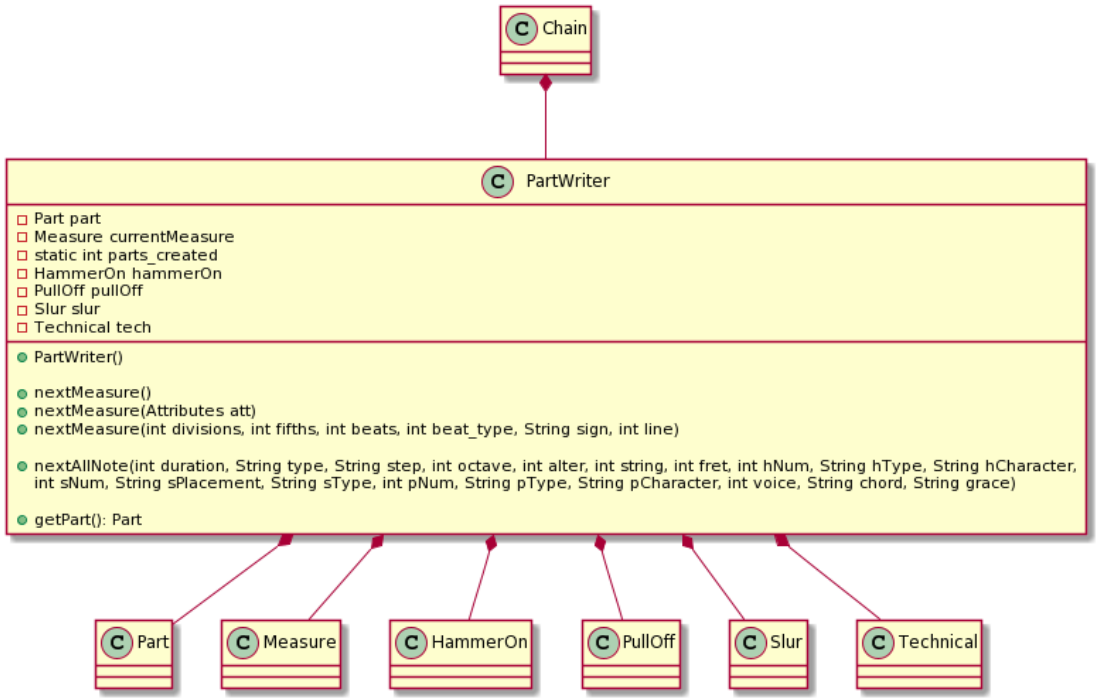
## *2.3 UML Class Diagram (Chain, XMLwriters)*



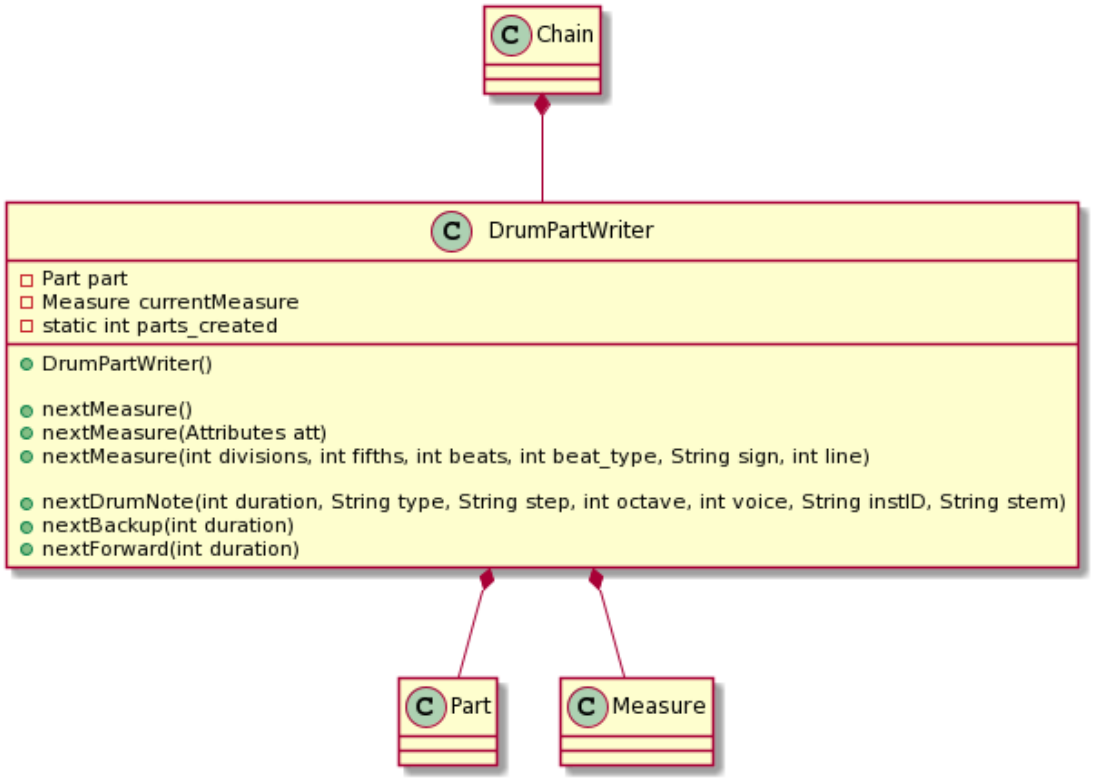

Chain holds all of the parsing and exporting data inside of it and helps move the data around to other objects. It is the assembly line of our program.



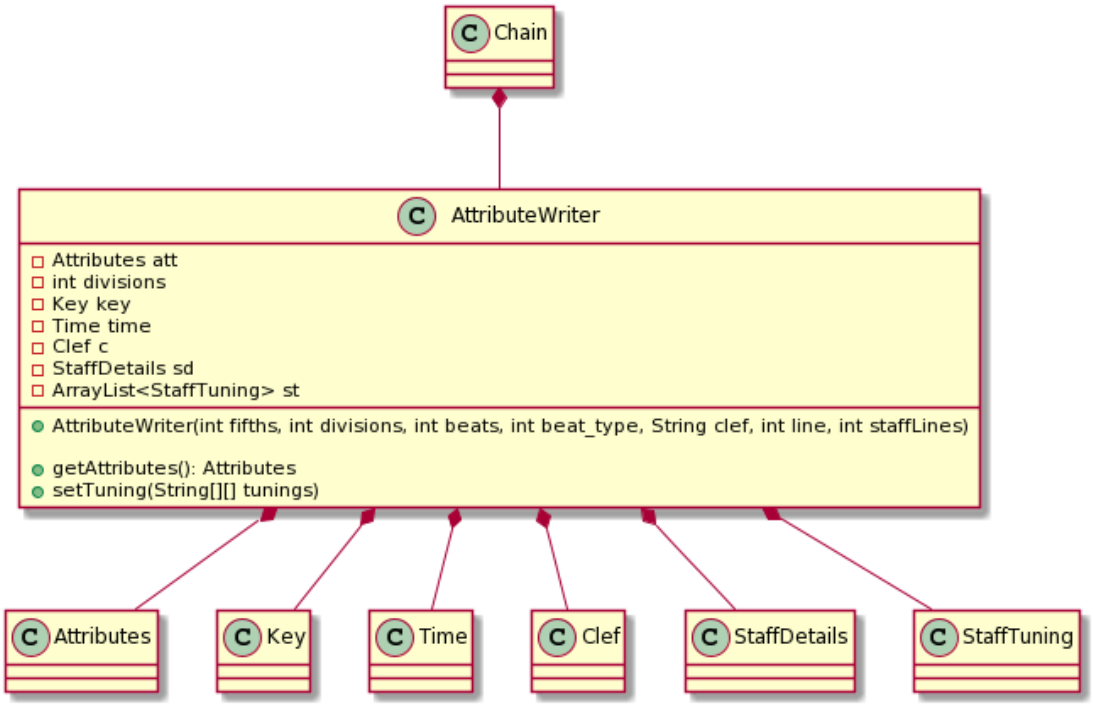

ScorePartwiseWriter is used to write the ScorePartwise object and contains useful methods to properly format the export to XML.




PartwiseWriter is used to write the PartWise object and contains useful methods to properly format the export to XML.



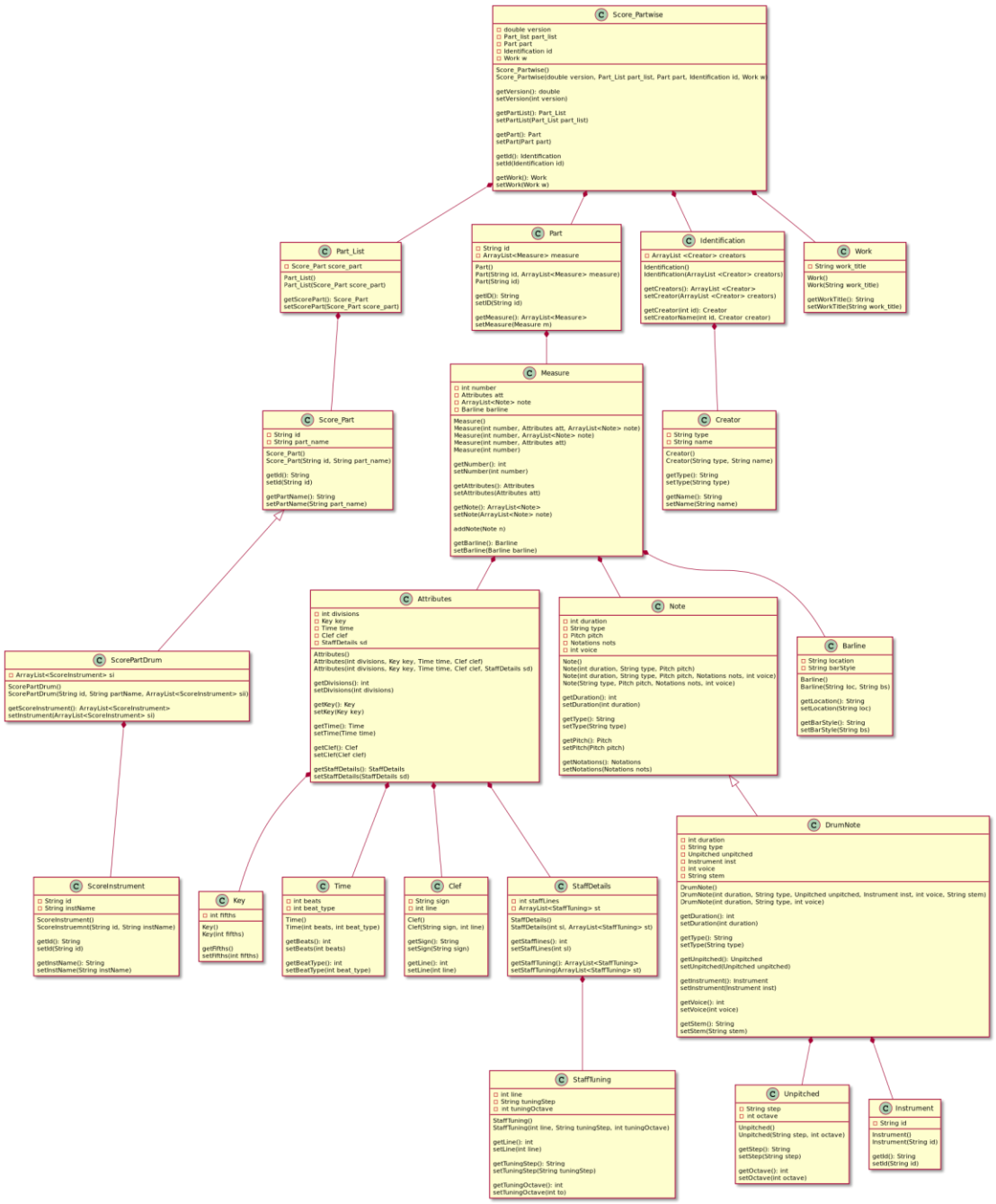

DrumPartWriter is used to write the DrumPart object and contains useful methods to properly format the export to XML.



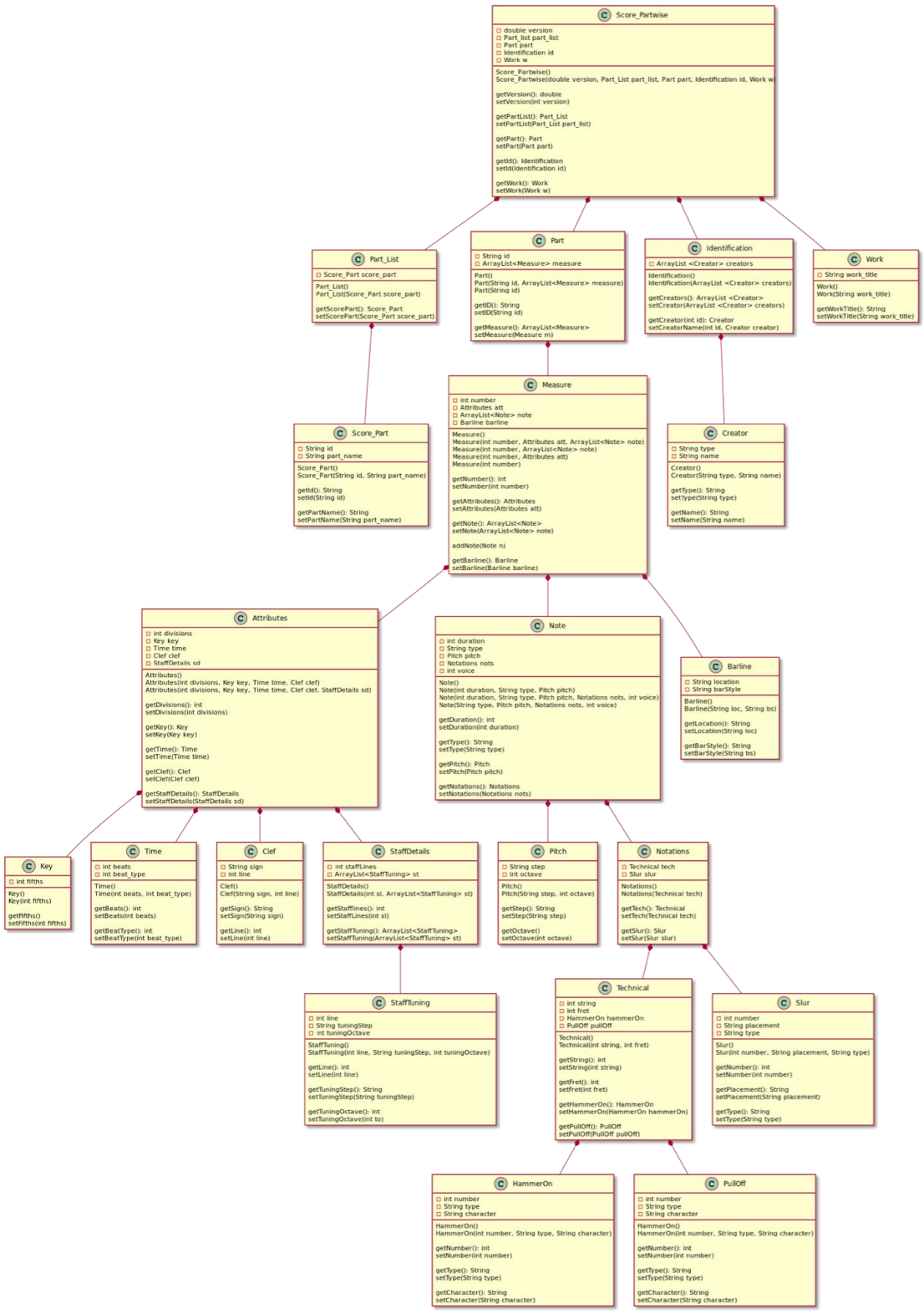
AttributeWriter is used to write the ScorePartise object and contains useful methods to properly format the export to XML.

## *2.4 UML Class Diagram (XML Classes, Drums)*

The XML Classes are the classes that hold the information for an XML element. In Java, there is a library called JAXB that is able to convert class objects into an XML file, this is the idea for XML Classes. The class diagram is designed in a hierarchical way that allows the output XML to be correctly formatted. Every "child" of a given class (composition arrows), becomes nested in its "parent" in the music XML. This class diagram essentially shows the correct format of a music XML, through class object composition. The content inside each class includes get and set methods, as well as variables; understanding these methods and variables are not important to understand the design. Rather, look at how the hierarchy is layed out to understand what the outputted music XML might look like. For example, the 'Work' class will be an XML element of 'Score_Partwise', and so on. The class diagrams for Drums and Guitar are relatively designed the same, with a few differences like Class names (Unpitched in drums vs Pitch in Guitar), as well as some extra classes (Drums need Backward and Forward classes, guitar does not).

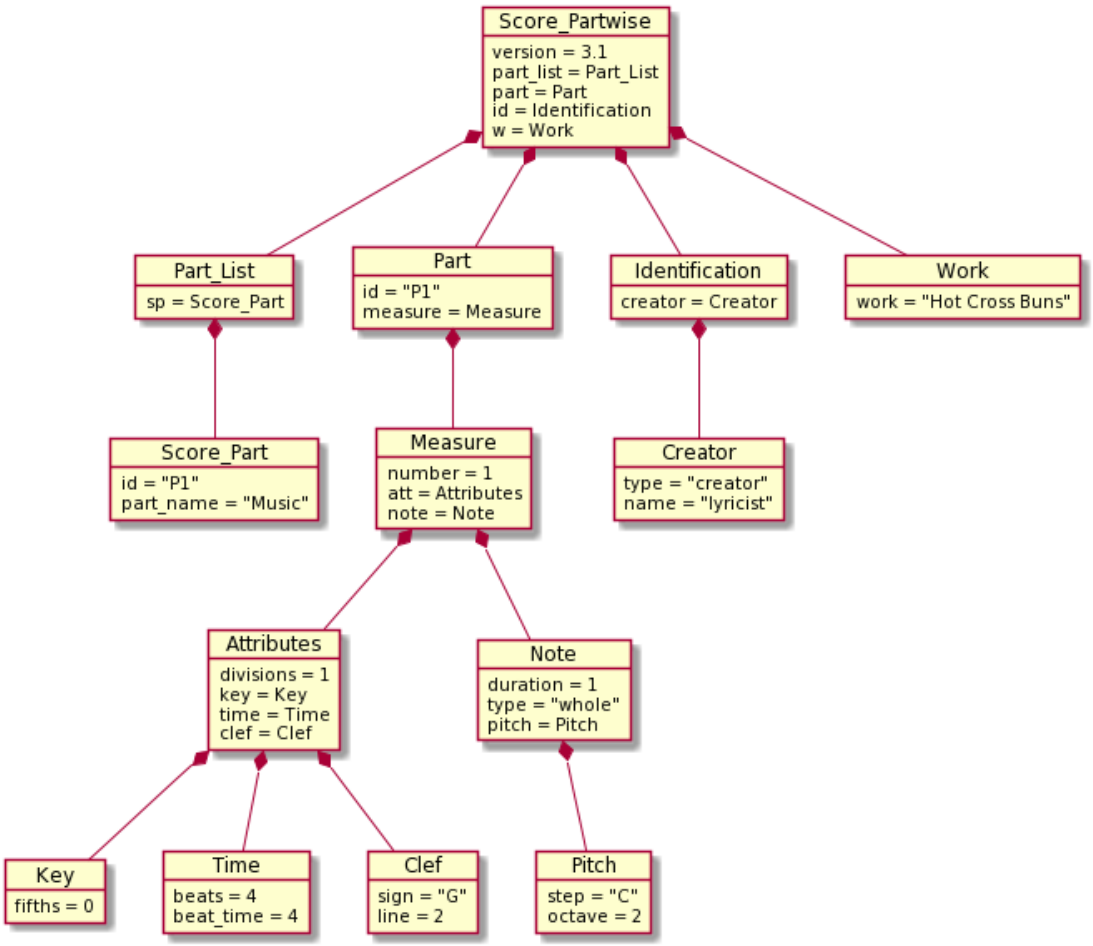

Score_Partwise
Part_List
Part
Identification
Work
Measure
Score_Part
Creator
Attributes
Note
Barline
ScorePartDrum
ScoreInstrument
Key
Time
Clef
StaffDetails
DrumNote
StaffTuning
Unpitched
Instrument

## *2.5 UML Class Diagram (XML Classes, Guitar)*

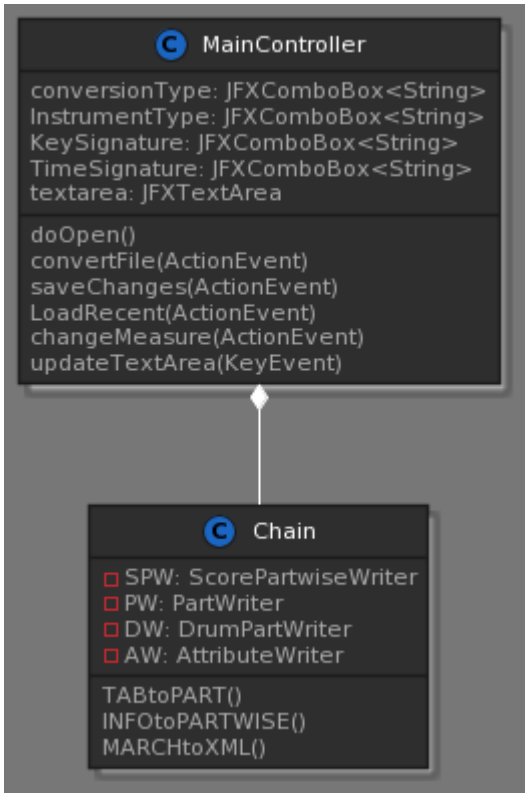


## *2.6 UML Object Diagram (XML Classes, Guitar)*

The following is a much more abstract view of the XML classes, it shows what a given class would contain (objects, variables). This should help further increase your understanding of what the hierarchy would look like when the objects are actually created in Java, as well as what the corresponding XML output might look like.



## *2.7 UML Object Diagram (GUI controller class)*



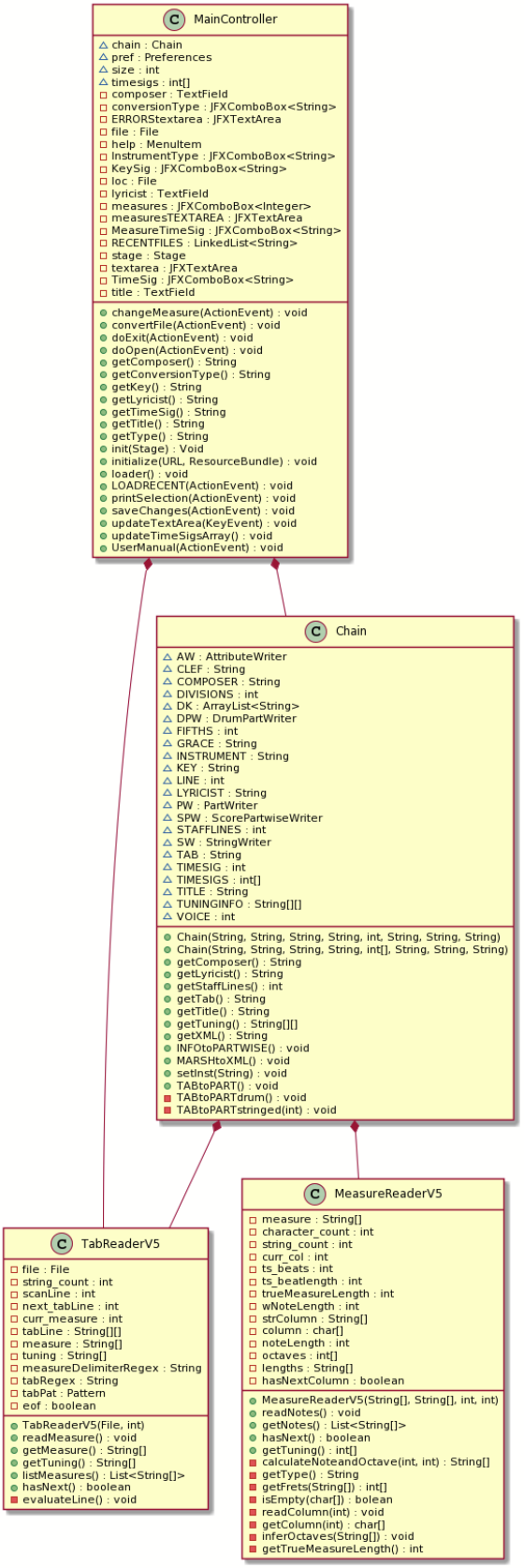

The main methods in the GUI class include:

| Method | Description/functionality |
|---|---|
| doOpen() | This method is invoked when the user click on the "Upload file" button, prompting the user to pick a *.txt file through a file chooser, and updates the text area (either appends the file or overwrites the current content according to the user's input). |
| convertFile() | This method is called when the user clicks on the "export" button. Mainly, creates a Chain object that takes in the tab input and all of its associated attributes, parses it (with the help of numerous XML classes). The method then prompts the user to specify a name and location of the converted tab through a file chooser and creates the MusicXML file. |
| saveChanges() | This method is called when the user clicks on the "save changes" button on the GUI. Its main functionality is to store the current |

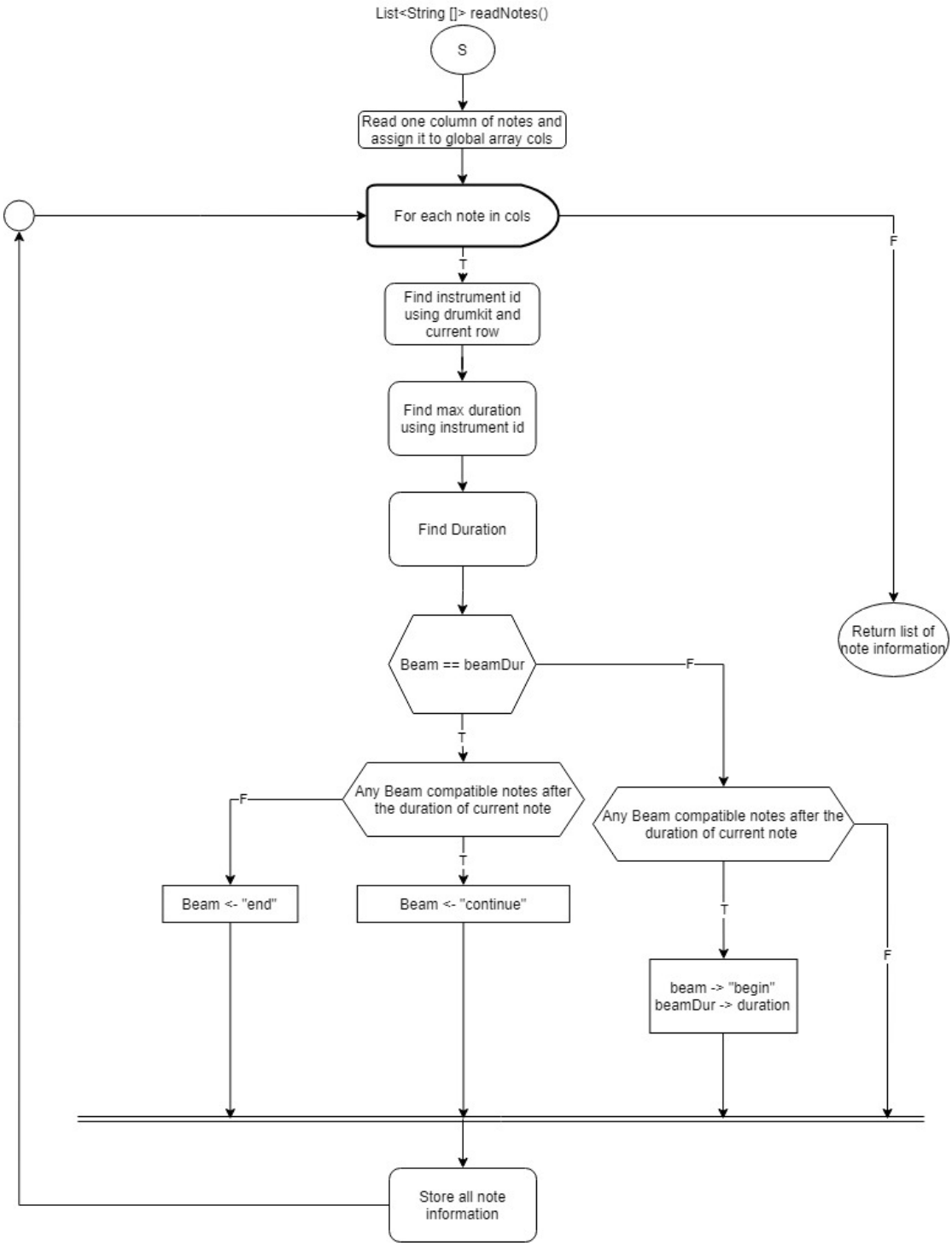
| | |
|---|---|
| | textarea input and all of its attributes to *.txt files that act as a database. |
| LoadRecent() | This method reads the content of two *.txt files (tab and attributes) and changes the GUI settings accordingly |
| changeMeasure() | This method is called every time the user picks a time signature to change one or more measures to. Its main functionality is to update the time signatures array for each measure individually. |
| updateTextArea() | The main functionality of this method is to update the textarea as changes/edits are being made to it, catch exceptions and append the appropriate error message in the GUI as to continuously warn the user of errors and warnings before exporting |

## *2.8 UML Object Diagram (Guitar Parser classes)*

Below is the object diagram for the guitar parser. It shows the interaction with the Chain class (which calls upon the parsing), and the MainController class (which is the GUI). It parses in 2 separate phases, separating measures at the Tab level, and Parsing individual notes at the measure level. This was done to be flexible, and allow shared use of the Tab level parser by the drum parser.


MainController
chain : Chain
pref : Preferences
size : int
timesigs : int[]
composer : TextField
conversionType : JFXComboBox<String>
ERRORStextarea : JFXTextArea
file : File
help : MenuItem
InstrumentType : JFXComboBox<String>
KeySig : JFXComboBox<String>
loc : File
lyricist : TextField
measures : JFXComboBox<Integer>
measuresTEXTAREA : JFXTextArea
MeasureTimeSig : JFXComboBox<String>
RECENTFILES : LinkedList<String>
stage : Stage
textarea : JFXTextArea
TimeSig : JFXComboBox<String>
title : TextField
changeMeasure(ActionEvent) : void
convertFile(ActionEvent) : void
doExit(ActionEvent) : void
doOpen(ActionEvent) : void
getComposer() : String
getConversionType() : String
getKey() : String
getLyricist() : String
getTimeSig() : String
getTitle() : String
getType() : String
init(Stage) : Void
initialize(URL, ResourceBundle) : void
loader() : void
LOADRECENT(ActionEvent) : void
printSelection(ActionEvent) : void
saveChanges(ActionEvent) : void
updateTextArea(KeyEvent) : void
updateTimeSigsArray() : void
UserManual(ActionEvent) : void
Chain
AW : AttributeWriter
CLEF : String
COMPOSER : String
DIVISIONS : int
DK : ArrayList<String>
DPW : DrumPartWriter
FIFTHS : int
GRACE : String
INSTRUMENT : String
KEY : String
LINE : int
LYRICIST : String
PW : PartWriter
SPW : ScorePartwiseWriter
STAFFLINES : int
SW : StringWriter
TAB : String
TIMESIG : int
TIMESIGS : int[]
TITLE : String
TUNINGINFO : String[][]
VOICE : int
Chain(String, String, String, String, int, String, String, String)
Chain(String, String, String, String, int[], String, String, String)
getComposer() : String
getLyricist() : String
getStaffLines() : int
getTab() : String
getTitle() : String
getTuning() : String[][]
getXML() : String
INFOtoPARTWISE() : void
MARSHtoXML() : void
setInst(String) : void
TABtoPART() : void
TABtoPARTdrum() : void
TABtoPARTstringed(int) : void
MeasureReaderV5
measure : String[]
character_count : int
string_count : int
curr_col : int
ts_beats : int
ts_beatlength : int
trueMeasureLength : int
wNoteLength : int
strColumn : String[]
column : char[]
noteLength : int
octaves : int[]
lengths : String[]
hasNextColumn : boolean
MeasureReaderV5(String[], String[], int, int)
readNotes() : void
getNotes() : List<String[]>
hasNext() : boolean
getTuning() : int[]
calculateNoteandOctave(int, int) : String[]
getType() : String
getFrets(String[]) : int[]
isEmpty(char[]) : bolean
readColumn(int) : void
getColumn(int) : char[]
inferOctaves(String[]) : void
getTrueMeasureLength() : int
TabReaderV5
file : File
string_count : int
scanLine : int
next_tabLine : int
curr_measure : int
tabLine : String[][]
measure : String[]
tuning : String[]
measureDelimiterRegex : String
tabRegex : String
tabPat : Pattern
eof : boolean
TabReaderV5(File, int)
readMeasure() : void
getMeasure() : String[]
getTuning() : String[]
listMeasures() : List<String[]>
hasNext() : boolean
evaluateLine() : void

## *2.9 UML Activity Diagram (DrumReader readNoteRow method)*




This diagram shows the readNotes() method in the DrumReader class, used to extract note data, from a string array of drum notes in a tab format. Since there are many different variations of input the algorithm to find this data

# *3 Maintenance Scenarios*

## *3.1 Adding Tab Features*

If a new tablature feature would need to be implemented in our system, here is how the general workflow for how it would be done.

1. Add the corresponding XML class, a Java class named “Vibrato” for example. Include the relevant XML elements and attributes, constructors and variables.
2. Add support for new character ‘v’ for vibrato in the parsing classes.
3. Add the new Vibrato object in the corresponding “Writer” class constructor for it to be properly instantiated.
4. Finally, modify the newly changed “Writer” object construction in the Chain class to accurately match the constructor that was changed. The chain class will invoke the parser, the parse will then pass the relevant information to chain (in this case it will pass in vibrato information now), and chain will create the XML.

After these steps, the newly added tab feature will be properly supported in our system. This can be applied for any new addition to the tab structure.

## *3.2 Adding GUI Features*

1. Adding features to the GUI has two main steps:
    a. Create and edit an fxml file in SceneBuilder
    b. Create a new FXML file under the resources folder
    c. Open the FXML in SceneBuilder
    d. Add JavaFX components to GUI
    e. Associate an ID and onAction method with each component as necessary
2. Implement the action events in the GUI Controller class
    a. Implement the onAction methods appropriately