Software Design Documentation

MusicXML Converter

EECS2311 Software Development Project

Date: 11/04/2021

Group 14

Adil Hashmi - adilhashmi.w@gmail.com

Alexander Arnold - alex290@my.yorku.ca

Boho Kim - kimboho614@gmail.com

Kanwarjot Bharaj - kjsingh76@gmail.com

Stanley Ihesiulo - stanihe1901@gmail.com

Table of Contents

1. Introduction

This document is a high-level structured overview of the design implementation in the MusicXML application, including important classes, methods, and how they interact which each other. This document will also include the objects created at runtime and how they are connected.

2. System Overview

The application is built and designed using Java, Fxml, and Gradle. Compatible with both Eclipse and IntelliJ. The software includes features such as a graphical user interface (GUI), easy to use with an intuitive design. A MusicXML file generator, and finally an end-to-end object-oriented translating algorithm. The algorithm translates tablatures that are user-inputted or entire .txt files. This software is designed for three stages with the first one being to get user input, then translating input and generating MusicXML, and finally creating a new file with the final product ready for use.

3. Design Considerations

Before creating a design solution, there are some factors that need to be considered, these include the Robustness of the design, Skill required, Resources, and the Time framework. Establishing these upfront helps understand the project as a whole, and aids in the planning process.

3.1. Assumptions and Dependencies

The user must understand music in general, there are features such as ‘beat-type’, ‘beat’, and many more that are essential to re-structuring musical tablatures in the way the user desires. The user must know what these do, and how they affect the sound of the music as a whole.

3.2. General Constraints

The main constraint this project deals with are Time, we are on a tight schedule as the due date for this project is coming up soon.

3.3. Goals and Guidelines

Our teams’ goal is to implement all three instruments with flawless performance by the due date. We also are looking to enhance user experience with better functioning error highlighting.

3.4. Development Method

The software design used *(figure 1).*

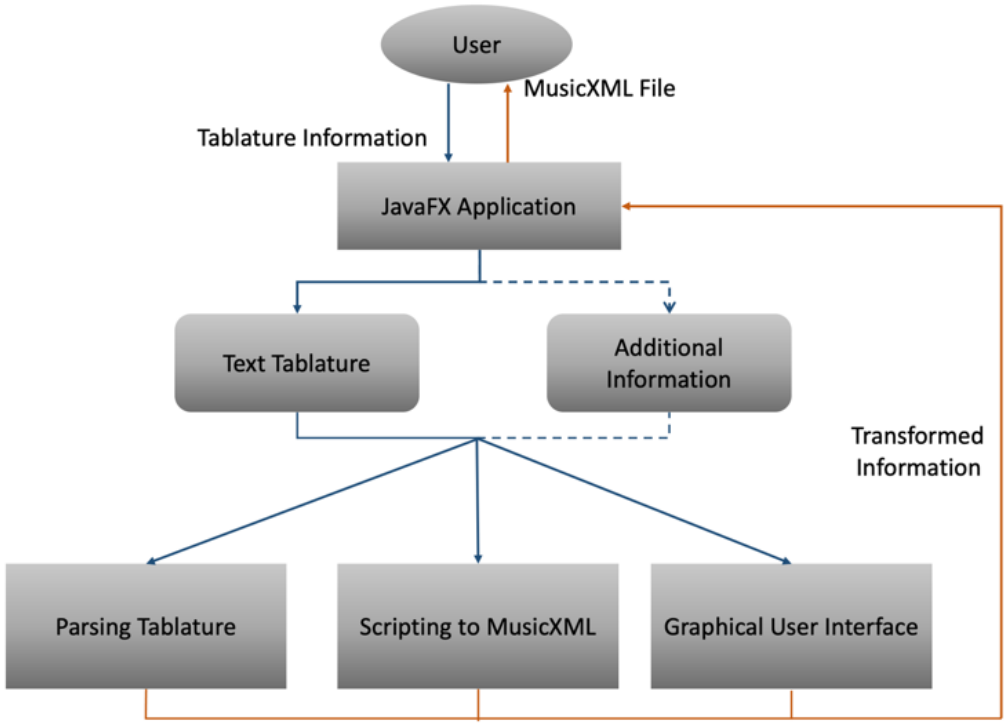

Figure 1

This software design is an easy and intuitive functional design benefiting the user. We focus on elegance, so there are minimal instructions and buttons because this way the system does all the work for the user, all the user needs to worry about is their music and how they want it to be structured, this way the user does not need to understand too much of how the technology works, which eliminates stress and creates more comfort and a better user experience.

This diagram below shows the Aggregation Hierarchy as well the Validation Algorithm diagram. It shows how the system validates each tablature to ensure that it is in fact a valid input, able to be parsed and recognizable by the error finder.
(figure 2)

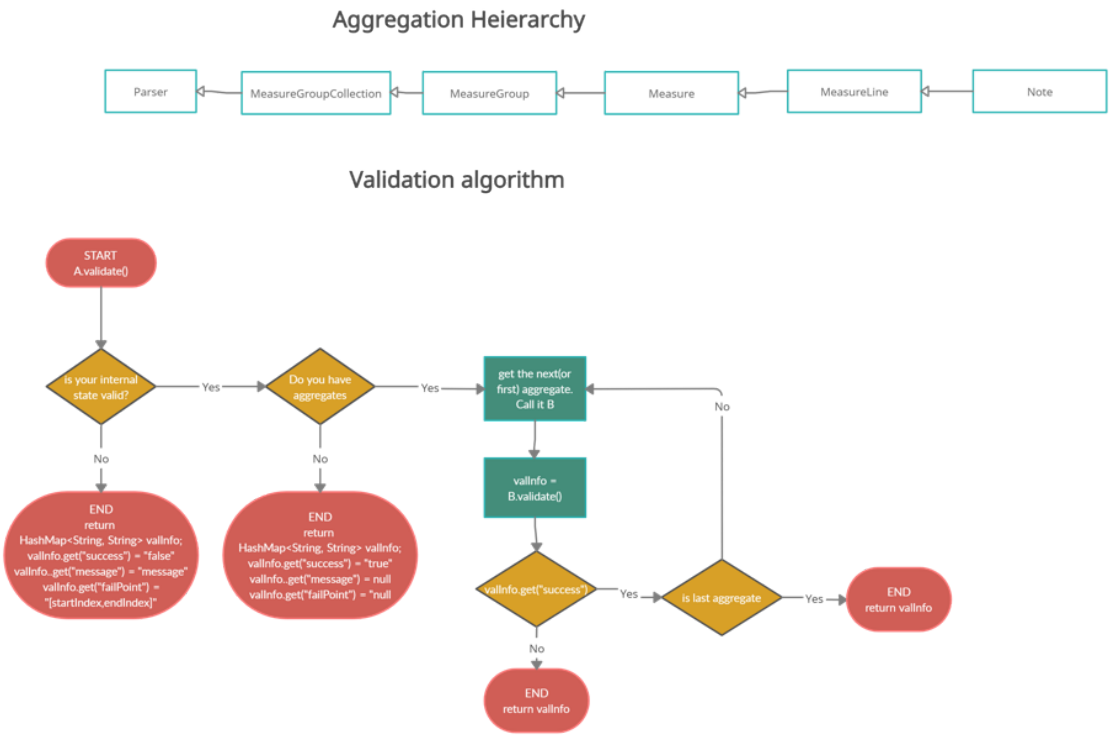

Figure 2

4. System Architecture

4.1. Important Classes

Score class

- Breaks down root string into string fragments wherever there is a newline so that regex parsing does not throw a stack overflow error.
- Finds, creates and stores measure collection objects from the string fragments.

MeasureCollection and MeasureGroup class'

- For each measure collection object: Creates and stores instructions and measure group objects from the origin string.
- For each measure group it creates and stores measure objects from its origin string for each measure object, it generates a list of measure lines from its origin string there are different types of the measure abstract class.
- The measure class is responsible for figuring out what type of measure a string represents *(i.e the type of instrument)* the abstract measure class also figures out the duration and the divisions for the notes.

Measure Class

- The abstract measure class also figures out the duration and the divisions for the notes.

GuitarMeasure and DrumMeasure class' (Concrete classes)

- The concrete measure classes are responsible for breaking down the origin string of the measure into measure lines.

Instruction class

- The instruction classes store their position relative to the nearest newline and when the instructions are applied to a measure collection, it only applies it to measures within the MeasureCollection which are within that range.

ScoreComponent Interface

- Each class implementing ScoreComponent interface has a validate method which uses the state of the class along with the index of its origin string to generate errors (stores as a list of HashMaps).
- All classes (except for instructions) that implement ScoreComponent have to have a getModel method.

5. Detailed System Design

5.1. Class Diagram

How the classes are designed at every state of building the XML *(figure 3).*

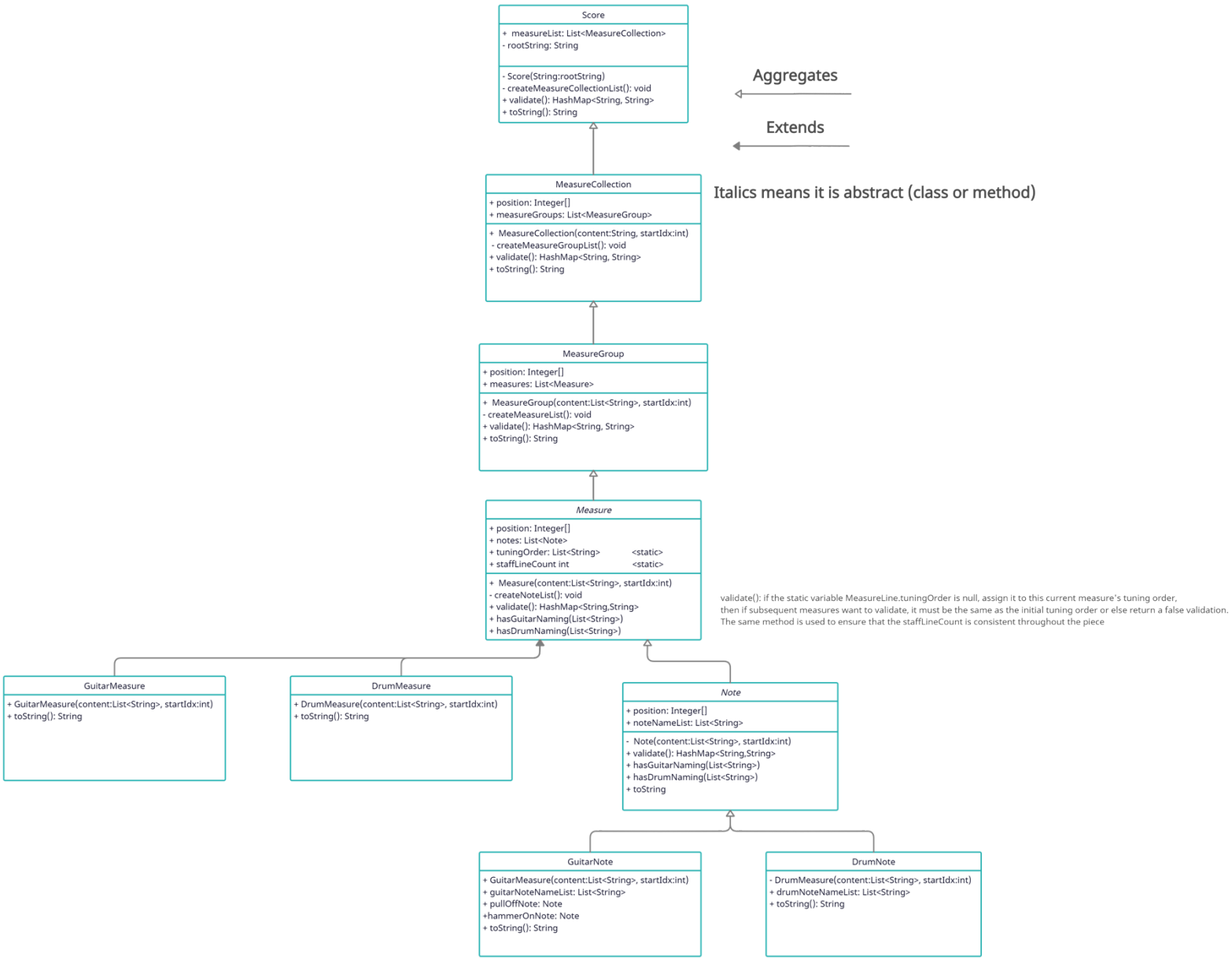

Figure 3

Example of how validate method works using GuitarConvert class diagram. *(figure 4)*

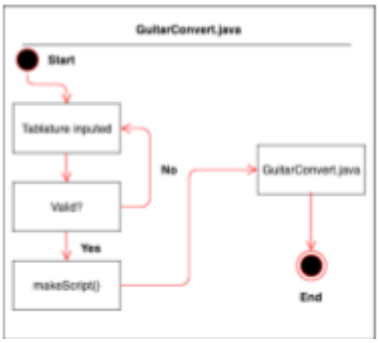

Figure 4

5.2. Sequence Diagram

The sequence diagram *(figure 5)* demonstrates how each class works injunction with one another.

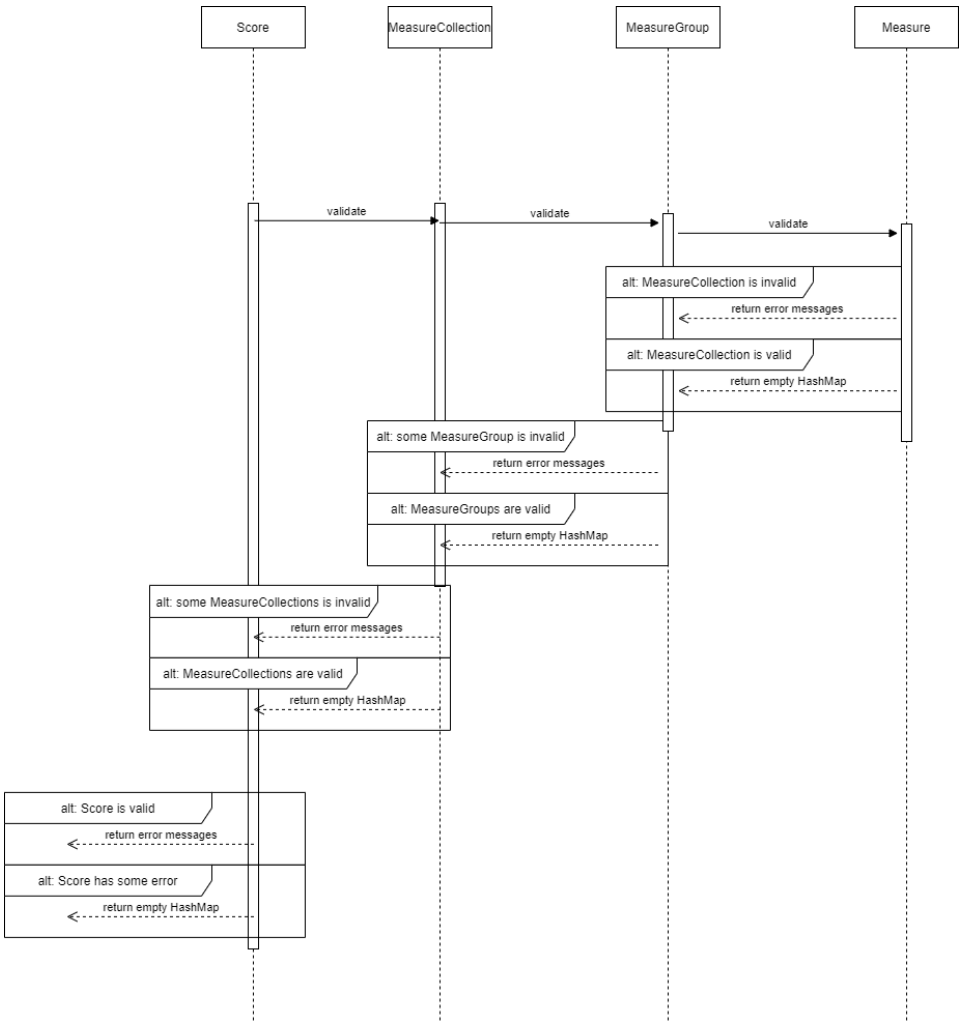

Figure 5

5.3. Runtime Objects

Objects that are created at runtime are JavaFX, RichTextFX, and Gradle (Although not a runtime object) contributes to running the entirety of the system for easy use.

6. Maintenance Scenarios

6.1. Supporting new Features

New additions of ease-of-use were added to enhance the user experience, the ‘Score Type’ is a new feature added to let the user select the type of instrument of their choosing with ease, or the user can select ‘Auto’ and let the system do the work. This was created in the *NotePlayer class.* Another new feature is the ‘Go to Measure’, this allows the user to edit any single measure of their choosing without the hassle of searching, the user can type in the measure number and the system takes the user to the exact line the measure is located. This was created in the *TabInput class*.

(Figure 6) shows the addition of the features on the bottom left of the user interface.

Figure 6

6.2. Adding new Drums and Notes

We added the ability to easily add new drum parts just by adding new entries into the text files located in src/main/resources/utility to specify details such as the display step, display octave, drum part id, etc of the new drum part. The logic of adding these new parts based on the CSV file is located in the filesrc/main/java/utility/DrumUtils.java

We also added the ability to easily add new note types by editing thesrc/main/java/converter/note/NoteFactory.java class to specify the appropriate regex for the note and specify a lambda function which would inject a change into the JacksonXML model for the musicxml to accurately convert the note into musicxml.

7. Error Considerations and Handling

7.1. Errors that we handle

1. When there are a mix of guitar measures and drum measures in a measure group.
2. When there are a mix of guitar measure line names and drum measures in a measure.
3. When measures in a measure group have different number of measure lines.
4. When there are unrecognized notes.
5. When there are unrecognized note names.

7.2. Warnings that we handle

6. Warning when the lines of a measure are unequal in length.
7. When a guitar measure doesn't have 6 measure lines.
8. Warning when whitespace is found in a measure line.

(May lead to a different duration than the user expects)

9. When a note is invalid.
10. When a hammer on goes from a higher note to a lower note.
11. When a pull off goes from a lower to a higher note.
12. When slide up goes from higher to lower note.
13. When slide down goes from lower to upper note.