TAB2XML Design Document

For version 0.4.0

2021 March 26

Contents

1 Front End Design

All TAB2XML front-end code is located in the `tab2xml.gui` *package.*

1.1 Front End Classes

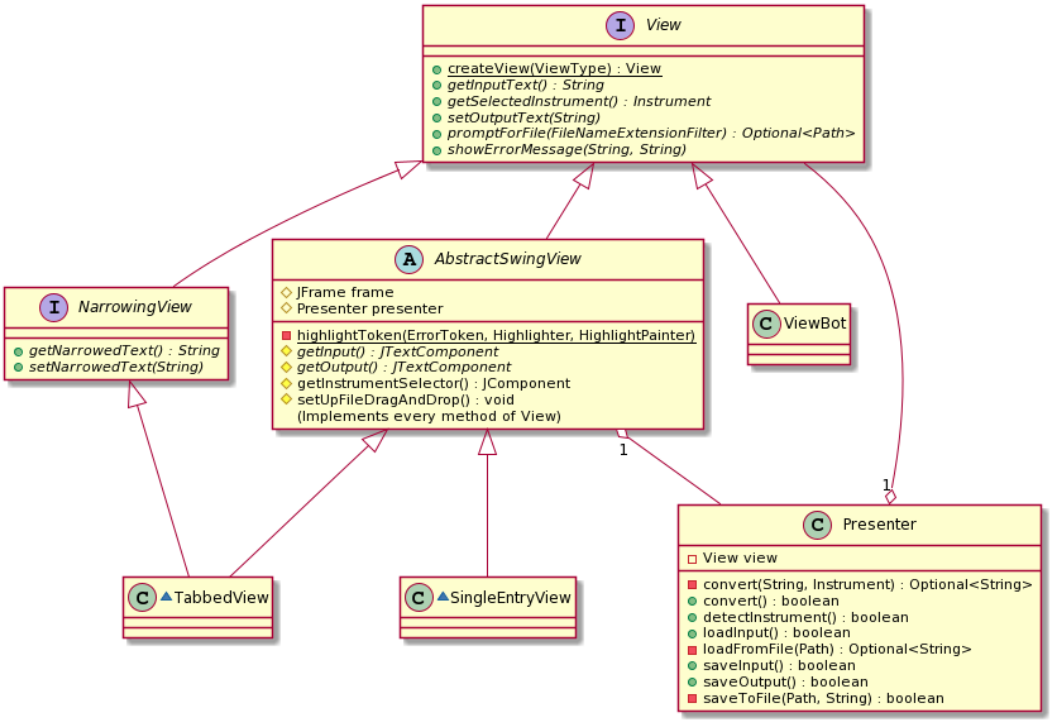

Figure 1: A class diagram for the frontend of TAB2XML.

The frontend of TAB2XML is designed using the MVP paradigm. It is divided into two main parts, the `View` and the `Presenter`.

The `View` is the part of the frontend that interacts with the user (the GUI). It is handled by the `View` interface; the GUIs for TAB2XML implement the View interface. In addition, all Views that represent a Swing GUI are subclasses of the skeletal implementation `AbsractSwingView`, which reduces the effort needed to make a View. Currently, there are four concrete classes implementing `View`: `TabbedView` (currently the one in use), `SingleEntryView`, `DoubleEntryView` and `ViewBot` (a mock view used for testing). The `NarrowingView` interface represents Views that additionally support TAB2XML's tab-narrowing functionality.

The `Presenter` is the part of the frontend that interacts with the backend code. It is a single class, not an interface that has multiple implementations. It implements behaviours such as converting a tab, loading from a file and detecting the instrument of the input tab. It uses the View interface's public methods to interact with the view. This means that the View's buttons can simply be linked to call the Presenter's methods, instead of having to implement the method in the View. All of the Presenter's methods return either a `boolean` or an `Optional` to describe whether they succeeded or not.

The rationale behind this design is to reduce the effort involved in creating a new GUI. If it extends `AbstractSwingView`, creating a new View is as simple as making a "mockup" Swing GUI and implementing two trivial methods. This makes it easy to work with multiple GUIs at once (allowing the customer to choose which they prefer). This design was especially important in the beginning of development, because I could prototype different GUI ideas with the customer using fully functional applications.

1.2 Converting Text Tabs

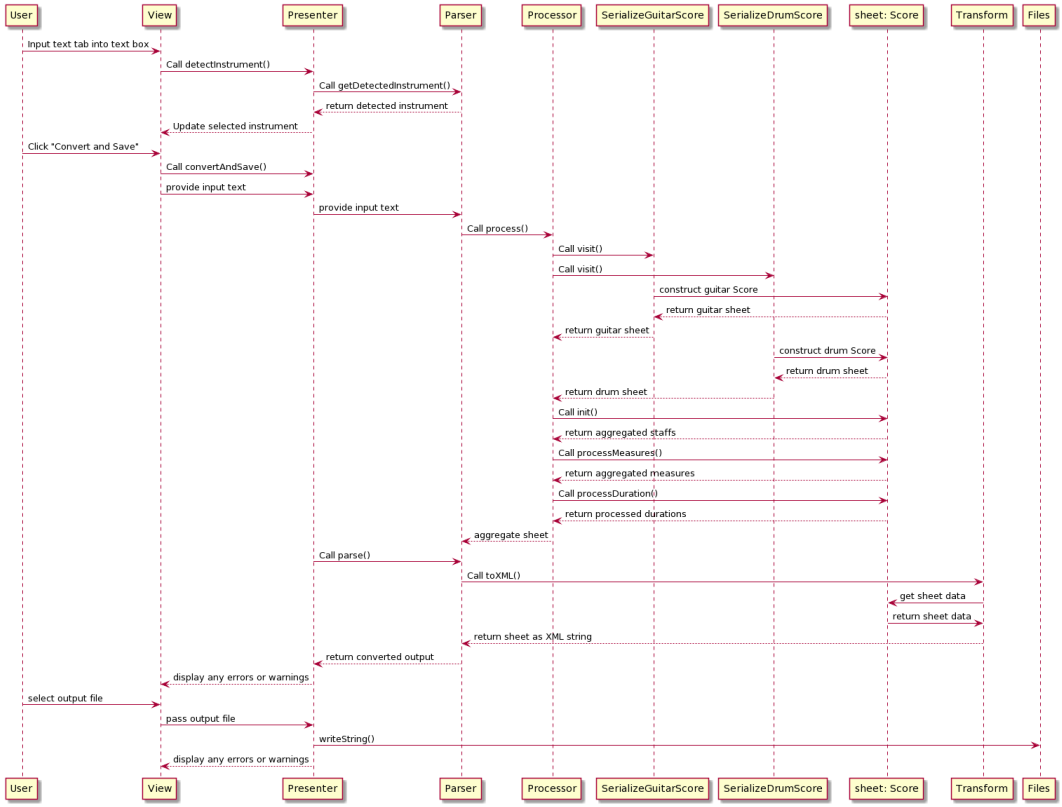

Figure 2: A sequence diagram for the "Convert and Save" operation

Here is how the "Convert and Save" operation works:

1. The user inputs the tab into the input text box (by typing, copy-and-pasting, the "Load from File" button or dragging and dropping a file).
2. The `View` calls the backend method `Parser.getDetectedInstrument(String)` with its text as input.
3. If it succeeded, the `View` sets its selected instrument to the detected instrument.
4. The user clicks the "Convert and Save" button.
5. The `View` calls the `Presenter`'s `convertAndSave()` method.

6. The `Presenter` calls the `View`'s `getInputText()` and `getSelectedInstrument()` methods to get the input tab and selected instrument.

7. The `Presenter` creates a new instance of `Parser` with the obtained input text and instrument. It then calls the `Parser`'s `parse()` method to convert the text tab.

8. The `Parser` returns the MusicXML, as well as any errors that occurred. Critical errors are thrown as Exceptions, noncritical errors are returned. This distinction exists so that critical errors stop the parsing, while noncritical errors do not stop it.

9. The `View` displays any errors or warnings to the user.

10. The `Presenter` calls the `View`'s `promptForFile` method to prompt the user for the desired destination file.

11. The `Presenter` calls `Files.writeString` to write the text tab to the selected file.

12. The `View` displays any errors that occurred during the file-saving operation.

1.3 Front End Maintenance

To create a new GUI, simply make your GUI handled by a class that extends `View`. It should also have a `Presenter` field instantiated using `new Presenter(this)`. You must implement all of the `View`'s methods, which is much easier if you extend `AbstractSwingView`. Then, you can call the presenter's methods within the GUI, and your GUI will be fully connected to the TAB2XML backend.

To modify the look of an existing View such as `TabbedView` (or add/remove components), simply modify its constructor (you may have to edit the other methods, if they are broken by the change). If you are adding a new feature that should exist in every Swing View, consider instead adding it to `AbstractSwingView`, as this will make it available for every View.

2 Back End Design

2.1 Overview

The backend of TAB2XML was designed with the main focus of flexibility, and future scaling of the system. The central component of the system is the antlr4 parser generation tool. The system uses custom instrument defined grammar to recognize different formats of tablature. Since the system's grammar can be changed effortlessly this makes extending for different types of input much easier. With the combination of the generated antlr4 parser classes(located in `/src/generated/java`) and the system's custom model data classes, a tablature score can be abstracted into components which make handling the data simpler. The backend is divided into a three step process, the `preprocessing` of the tablature, the antlr4 `ParseTree` visitor which is used to extract score data, and finally the XML conversion process.

2.2 Model Design

All TAB2XML model code is located in the `tab2xml.model` package and its subsets `tab2xml.model.guitar` etc..

The design of the instrument based model classes have a one-to-one correspondence between the respective grammar. The system abstracts some of these components which are shared in all the tablature formats(Such as `Score`, `Staff`, `Note` objects). The `tab2xml.model` package contains general classes along with abstract data classes. In the model package, subsets `tab2xml.model.guitar` and `tab2xml.model.drum` are specific to the respective instruments. For example, a drum model will not contain a `Tune` representation and conversely a guitar model will not contain a `DrumType` representation.

2.2.1 Score object

The `Score` object is by far the most important part of the model as it contains all the other objects. Because of this, the system is designed to allow the `Score` to be essentially a custom data structure. With functions such as adding staffs, iterating over staffs, iterating over notes and adding measures. One of the most important parts in designing this system for the `Score` object was to make sure that the notes had a natural ordering. This would allow notes to be compared, sorted, and provide notes a positioning system. To achieve this, a custom iterator was defined along with the `Note` object being `Comparable`. This method of abstraction of the score has a lot of benefits during the conversion tablature conversion process.

2.2.2 Model Abstraction

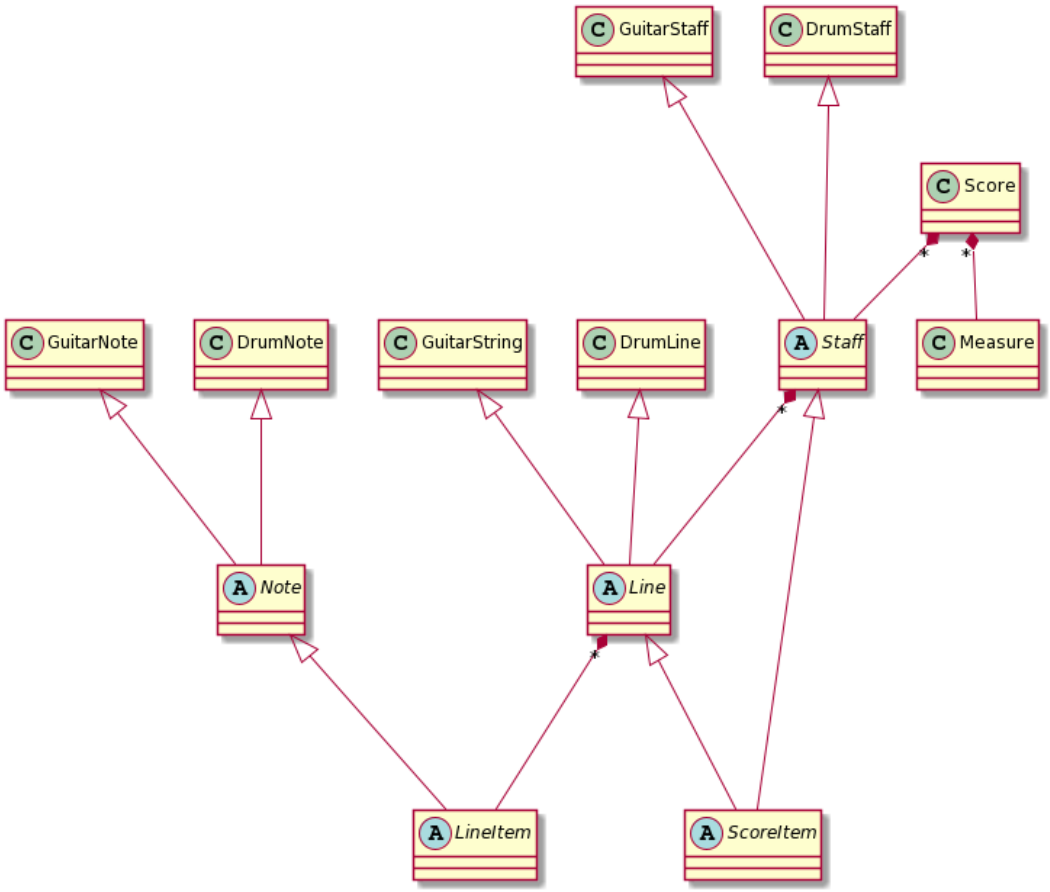

Figure 3: A general model diagram of the abstraction of a `Score` object.

2.3 Parser Design:

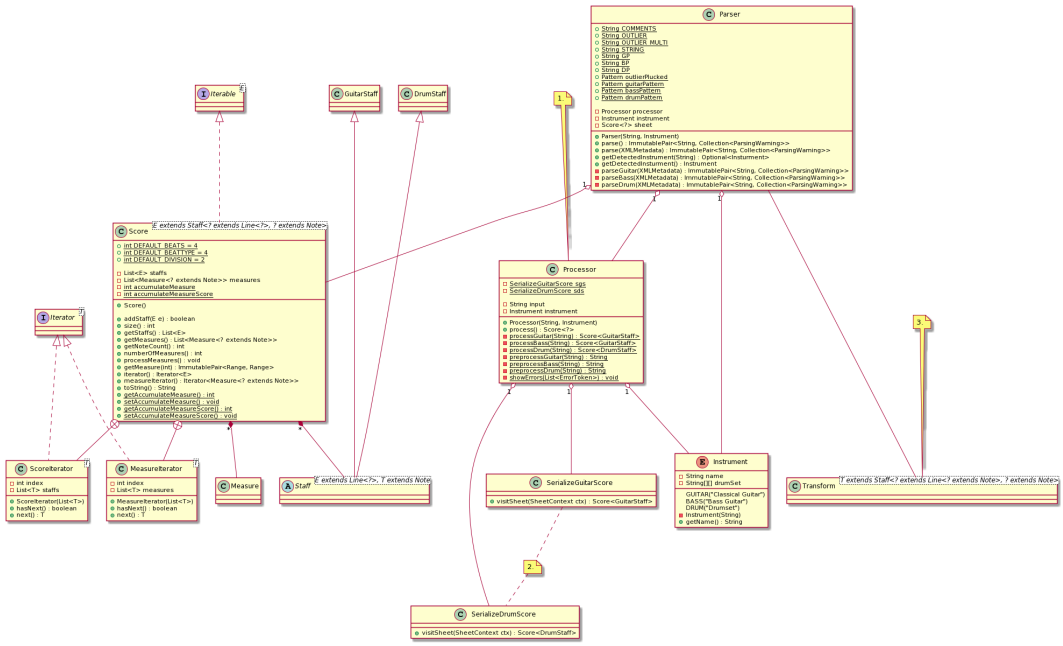

Figure 4: A class diagram for the `Parser` class.

The highlighted areas are the main components of the three main steps in the systems process as mentioned earlier. The first is the `Processor` which is aggregated with the `Parser`. The responsibility of the `Parser` is to unite the `Processor` and the `Transform` components and delegate conversions of tablature based on selected instrument or detected instrument. The `Processor` preprocesses the input to prepare it for the `ParseTree` extraction process. One of its preprocess tasks is to comment the metadata around the detected staffs in the score(The grammars are defined to ignore the commented metadata, although we still extract it as it might be useful to the user).

2.3.1 sample `Processor` task:

before preprocessing:

```
              III.......
  |       |       |      :  |       |       |
E|--0----------------------|-------------------------|
B|-----------------3-----5-|-2----------------------|
G|-----------------3-------|-2----------------------|
D|-----------------5-------|-2----------------------|
A|-------------------------|-0----------------------|
D|-------------------------|-------------------------|
                  3     4   1
```

after preprocessing:

```
/*
                  III.......
  |       |       |      :  |       |       |
*/
E|--0----------------------|-------------------------|
B|------------------3-----5-|-2----------------------|
G|------------------3-------|-2----------------------|
D|------------------5-------|-2----------------------|
A|--------------------------|-0----------------------|
D|--------------------------|-------------------------|
/*
                   3     4   1
*/
```

Once the main preprocessing tasks are complete and we are confident the input is valid, the `Processor` uses its aggregate extractor classes(ie. `SerializeGuitarScore`, `SerializeDrumScore`) to visit the parse tree generated by antlr4, while using the respective model classes to contain the information. The main steps of making the extracted data useful happens during the last steps of the `Processor`. Tasks such as creating measures for the `Score`, and calculating duration of notes. Once the processor has finished its job we have a `Score` object ready to be transformed into its XML equivalent. This is where The `Transform` class comes in. It's job is to simply generate XML from the parsed information serialized in the respective `Score` object. Hence, once this conversion is finished the XML is passed back to the frontend where it is handled as needed.

2.4 Grammar Design

The grammars for the system are designed to abstract the score representation. The grammars can be located at `src/main/antlr`. The system defines a set of rules for the grammar and antlr4 then creates a corresponding `ParseTree` from the input stream. The following are example rules(lower case, which would be nodes in the tree, sheet being the top level rule) and tokens which help build the grammar rules. This makes adding new support for tabs fairly easy as all you need to do is change the grammar rules and have a corresponding data model for that feature.

With the rules and `ParseTree` defined by antlr4, the system can traverse the `ParseTree` with the system's custom made `Visitor` classes(`SerializeGuitarScore`, `SerializeDrumScore`). The visitors define their logic in parsing the information based on which node the visitor is at in the input stream. If a hammer-on rule is reached the information is stored in the respective `HammerOn` model class. The visitors are broken up into three abstract components that serializes the `Score`, serializes `Staff`, and finally collects line/string(`GuitarString`, `DrumLine`) items. These classes all extend their respective grammar defined `BaseVisitor` classes generated by antlr4.

2.5 System capabilities

The system can support well formed guitar and bass tablature very well. The system's auto instrument detection is robust as it takes into account the metadata around the main com-

GuitarTab

Rules

- sheet
- staff
- string
- stringItems
- fret
- harmonic
- pulloff
- hammeron
- slide
- hampullchain
- tune
- NOTE
- FRET_NUM
- DOUBLEBAR
- BAR
- HYPHEN
- SPACE

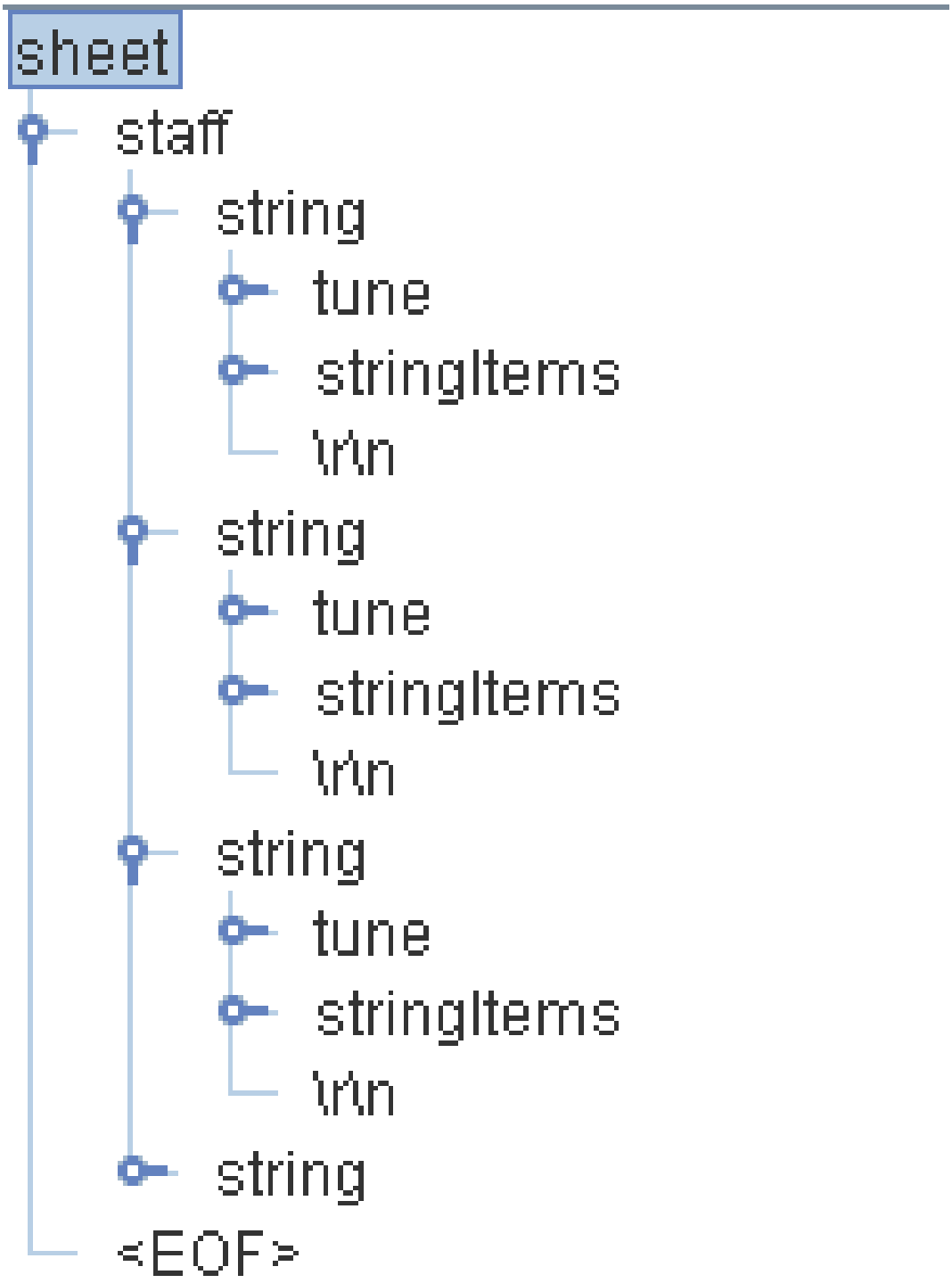

Figure 6: An example of a `ParseTree` structure for a bass(which is uses `GuitarTab.g4` grammar) tablature with four strings:

ponents of the score, making it convenient for the user. The system does fall short when the input is not well formed due to a lacking input validation system which allows malformed input to bypass to the `ParseTree` visitor process. With the right implementation and design of a validation system this could be fixed rather easily. The grammars of this system could also be improved to further reduce ambiguities which arise errors. The system's design abstraction of the `Score` object into its subcomponents extends the possibility to allow more detailed configuration as desired by the user.

2.6 Back End Maintenance

To add new support for a tablature feature you must change the grammar for the respective instrument. Adding a new rule is very simple but the main challenge is creating a grammar that avoids ambiguity. That's why it's important for the system to abstract the `Score` into subcomponents. For example, our system doesn't support bend actions for guitar. We can add this support by adding a rule `bend` in our grammar file and finally add that rule to our `stringItems` rule. Then finally parsing the information once that rule is reached in the `ParseTree`. This ease of changing the grammar makes it easy to extend support. The grammars are not perfect but it is a good base to extend to more complex features. The model classes all contain modular abstractions of classes which make them easy to maintain or add additional changes to. There is a clear distinction of class separation since our model is divided based on the respective instrument. Making it simple to create new models for currently supported instruments or ones we want to support.