Final Design Document

Team: Eric Ruleman (eruleman), Nicholas Mizoguchi (nickmm), Victor Horta (vhorta)

TA: Nils Molina

Design Revision

We had two preliminary design documents before we started coding. We reviewed our first one with our TA, and later on, when implementing our code, we figured out other changes that made our strategy clearer. Here are the main modifications made on the original design milestone document:

- *We abandoned the Section class.* Initially, our plans included creating a class to group all measures inside the same section on a specific instance of a Section class. However, this class wouldn't give us any extra expressivity, acting like another layer to forward and regroup instances of the Measure class. Given that, we abandoned the the Section class, and grouped measures under an instance of Voice.

- *Lyrics special characters are being treated directly as a String.* Using the Stack to handle special characters (such as "\-" and "_") by pushing and popping showed to be very complicated, and often gave us errors. We then choosed to retrieve a String corresponding to one line of lyrics from the abc file (when we exit the "lyric" parse rule) to treat and create all syllables as new Strings, and then push a fresh new List<String> to the stack.

- *We gathered all classes and control methods (including play) inside a KaraokeController object*. This way, we can clearly separate and encapsulate our karaoke player, and all the final user (represented as the Main class) needs to do is to instantiate a KaraokeController object by giving the path to the abc files folder -- the rest of the job is left for us.

- *We created a SequencerInformation class.* This class has the required information to create a singable note (syllable + sound information), and acts as an intermediate between the SequencePlayer and our ABCMusic object. We were previously planning to build a PreMidiNote class, but it was changed. We also created a class called MidiNoteRepresentation, that corresponds to a pitch, a start tick and a duration (also measured in ticks).

Testing strategy

Our test suite consists of a set of methods that cover the partitioned domain of each class. For every designed class, we have a correspondent JUnit test class. We adopted the Test-first programming strategy, so that all tests, rep invariants, documentations and class descriptions were written before the proper coding of the class. This way, our debugging became much easier, as our code was able to fail fast.

Our most general tests are at the ABCMusicTest class. We created five different abc files, each one containing a different error:

test_bad1.abc: Missing required header fields;
test_bad2.abc: Wrong header info given ;
test_bad3.abc: Invalid notes as inputs;
test_bad4.abc: Malformed repetition (without corresponding barlines);
test_bad5.abc: Voice and lyrics on non-standard positions.

We are checking if the expected exceptions are being thrown when trying to pass these files to the ABCMusic constructor.

We have also created personalized songs (from scratch!) to be tested:

payphone.abc: Maroon 5 - Payphone. Tests optional header fields;
summer_nights.abc: Grease - Summer nigths. Tests multiple voices;
street_fighter.abc: The intro theme from the Street Fighter II game (Capcom). It is being used to test valid repeats.

Many other tests were created, and their strategies can be found documented on the top of the respective .java file.

Classes diagram

The following classes diagram summarizes our final project design:

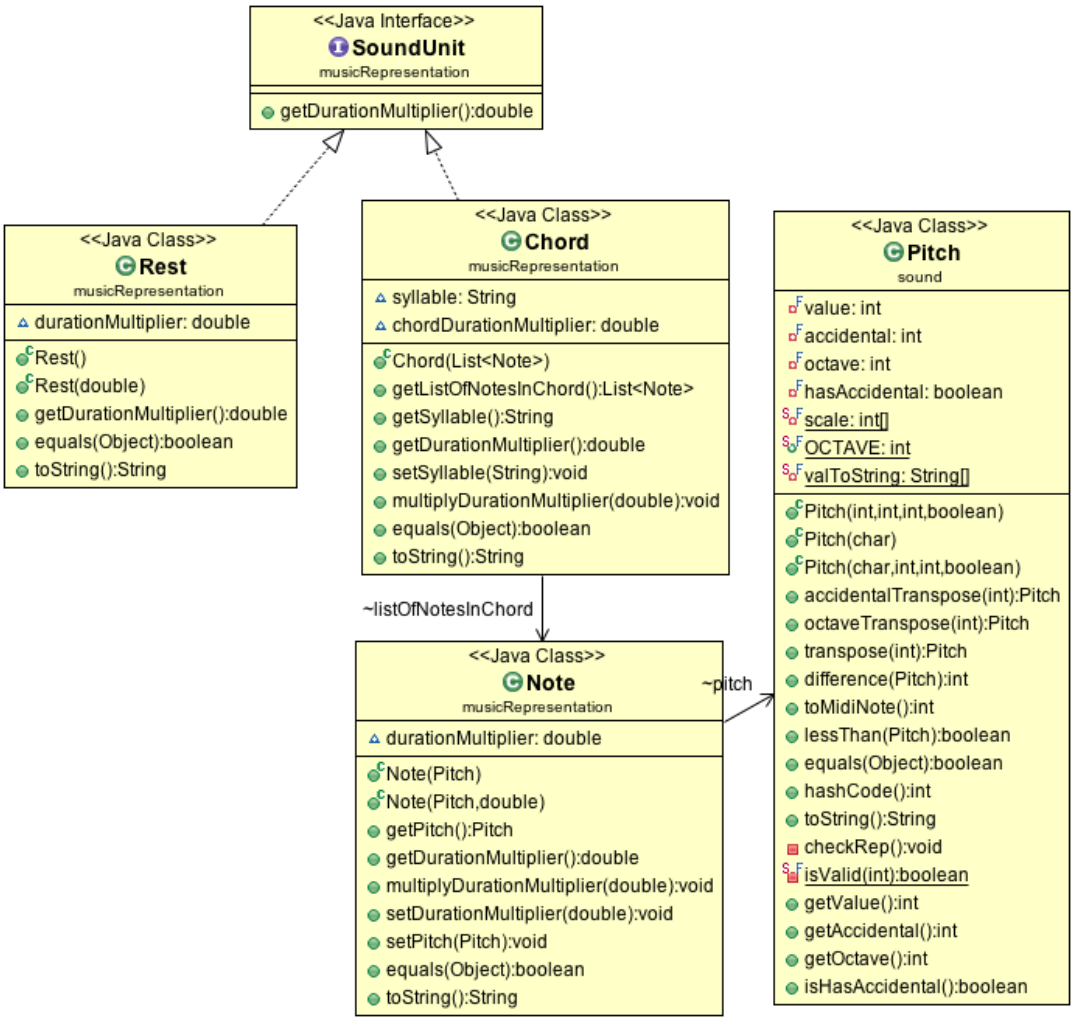

Figure 1 - SoundUnit Interface design

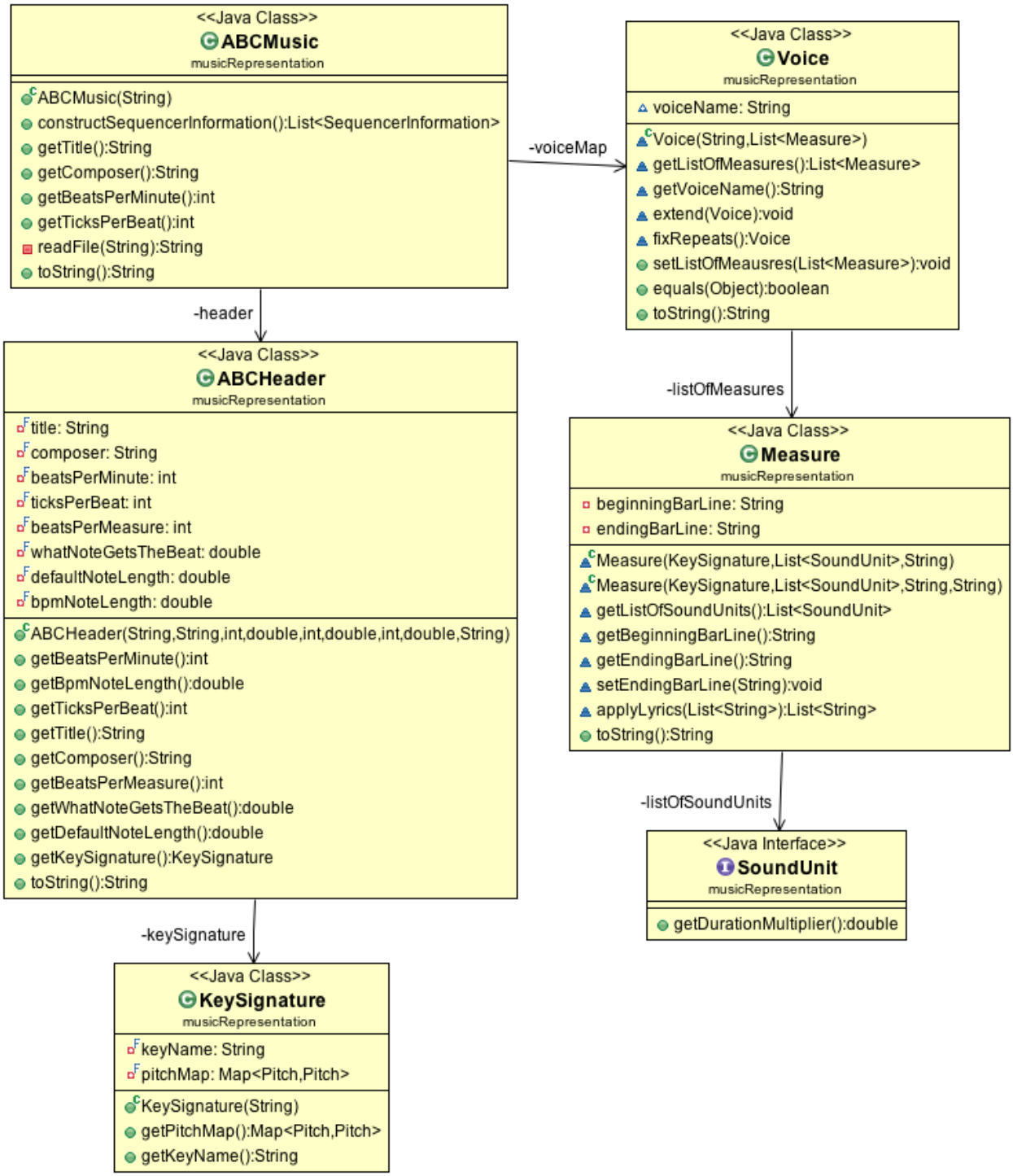

Figure 2 - ABCMusic class diagram

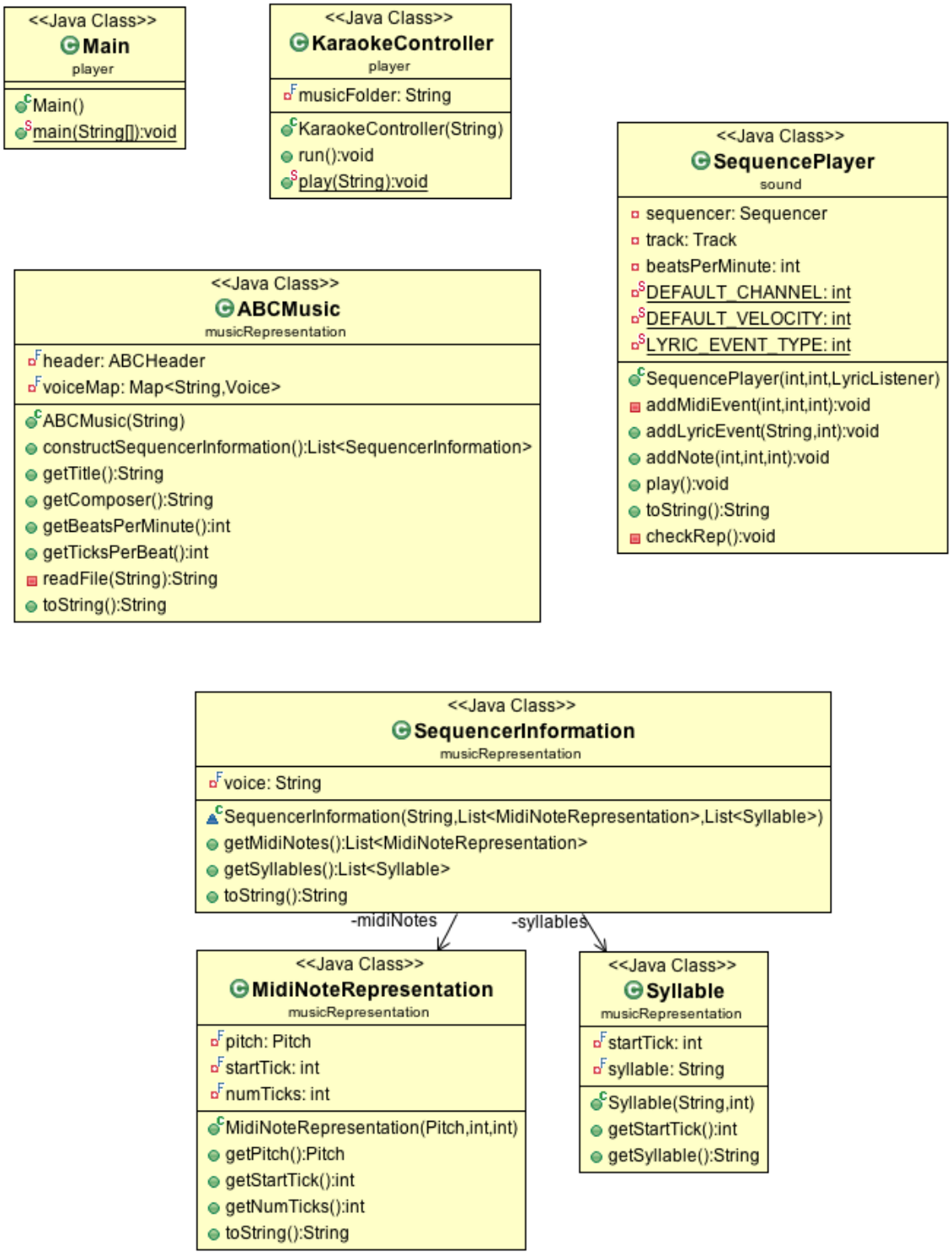

Figure 3 - Project class diagram