SOUNDLIB: A MUSIC LIBRARY FOR A NOVICE JAVA PROGRAMMER

Viera K. Proulx
College of Computer and Information Science
Northeastern University
Boston, MA 02115
617-373-2225
vkp@ccs.neu.edu

ABSTRACT

We describe the design, pedagogy, and student's experiences with a library that allows a novice Java programmer to design sound and musical accompaniment for interactive graphics-based games, as well as explore the programming of simple musical compositions, sound recordings, or visual representations of music and sound. The library has been used for three semesters in our classes and is publicly available at our website.

We wish to highlight two aspects of the library. First, the library explicitly supports our test-first design approach to teaching object-oriented programming. Second, the context of musical sounds: notes, pitches, duration, instruments, how they create a melody, how they can be represented in a number of different ways, presents a unique design playground for practicing class-based design.

INTRODUCTION

Introductory programming is hard to teach. Our goal is to give the student an opportunity to design a class-based system of non-trivial complexity, yet simple enough to be manageable with only the basic programming skills the student has mastered. To support this type of design exploration we have used for a number of years libraries (named *draw* with variants *idraw* and *adraw*) that create a `Canvas` for simple drawing of shapes, and a `World` class that handles time events and key events (the View and the Control), with students designing the game behavior (the Model). Student's game code extends the `World` class by providing the `onKeyEvent`, `onTick` and `draw` methods and it starts the game play by invoking the `bigBang` method.

Our curriculum enforces test-first design. Students are taught to design unit tests for every method as soon as its signature and purpose are defined. The programming of the game behavior supports this pedagogy. Students can define tests that check whether the state of the world after a key event or a timer tick corresponds to the expected one.

This simple environment for designing complex games has served us well for a while, but then it began to lose its lustre. In our first course students use a series of functional languages with similar game libraries. (Indeed, these libraries have served as models for our Java libraries.) As the supporting software and the first course evolved, the games became more complex, the game libraries added sophisticated image

manipulation as well as a support for programming distributed client-server based games. Coming into the second course that introduces class-based design in object-oriented language (Java) the excitement for designing games with just a simple graphics waned. The first game, designed in a mutation-free sub-language of Java was still a challenge. When asked to design yet another game in a mutable style students would recycle old games and retrofit them into the new style adding little to their class design repertoire.

Inspired by the work of Erich Neuwirth who uses programmatic music composition (in a mini language for spreadsheets, and in Logo) for introducing computer science concepts to beginners, we designed a sound library for Java. This library provides fresh design opportunities, challenges, and possibilities --- very different from those the students have already seen. Additionally, the design of the library itself includes several interesting case studies in class-based design that we plan to leverage in our course.

We present this tool and our experiences with it as follows. The next section describes the design considerations for both the tunes package and the isworld package and highlights the key features, including a deliberate design for testability. We then illustrate the use of the library through our demo program, as well as through several student projects, and conclude with acknowledgements.

LIBRARY DESIGN

The **SoundLib** library consists of the *tunes* and the *isworld* packages. The *tunes* package implements the music/sound component and allows the programmer to compose programmatically musical sequences (melodies, sounds) and play them. The *isdraw* package extends the functionality of the original imperative *idraw* library by allowing the programmer to play notes and other sounds on each tick or in response to a specific key event. It also adds the methods for responding to mouse events.

The *tunes* package: Building the orchestra

The tunes package defines a **`MusicBox`** class that initializes a *MIDI* Synthesizer with the default `Soundbank` and defines the initial *MIDI* program change selecting 16 instruments that provide a variety of options for the student. There are methods that allow the programmer to see what is the current *MIDI* program (assignment of instruments to channels) and that allow the programmer to change the instrument assignments (the *MIDI* program). Additional methods allow the programmer to start and stop playing one or more `Tunes`, where a `Tune` represents a channel choice and a `Chord` (a collection of `Notes` to be played or stopped). A simple `sleepSome` method that allows the given time to pass before resuming completes this class. The programmer can play tunes by starting and stopping a sequence of tunes with pauses in between.

To make the *MIDI* musical notation accessible to the programmer and to allow students with minimal musical background to use the library, the interface **`SoundConstants`** defines names for all *MIDI* instruments (e.g. `PIANO`, `TUBA`, or `BIRDTWEET`), provides simple names to represent the notes in the middle of the piano

keyboard e.g. `noteC`, or `noteDownG`), and contains a mapping of *MIDI* instrument numbers to their names. This makes it possible to define a `Tune` as simply as:

```
Tune pianoA = new Tune(PIANO, new Note(NoteA));
```

Once a synthesizer has been initialized and the assignment of the instruments to the sixteen *MIDI* channel has been completed, the tunes to be played need to specify only the note and the instrument on which the note should play.

When designing a musical component for a game, all that student needs to do is to decide which notes or tunes should be played on each tick, or in response to the key event. Adding the chosen notes or tunes to the appropriate `TuneBucket` plays the selected melody. We carry the tunes in a bucket, so even the most musically challenged programmer could carry a tune :).

The **`Note`** class allows the programmer to define any of the 128 *MIDI* pitches with a selected duration in a number of different ways. The class is designed to accept a number of different formats for defining the note, supporting different types of users in a comprehensive manner. The programmer can specify just the pitch. The default duration is one tick. Of course, students may not want to remember that 60 represents the middle C, and so the note names defined in the `SoundConstants` interface provide an easy way out. So, the middle C note of duration 1 may be defined in any of the following ways:

```
Note midC = new Note(60);
Note midC = new Note(noteC);
Note midC = new Note(60, 1);
Note midC = new Note(noteC, 1);
Note midC = new Note("C4n1");
```

The last variant uses a `String` representation of the note, given by the note name `C`, the octave `4`, the modifier (one of `n` natural, `s` sharp, or `f` flat), and duration (1 tick). The design of the constructors in this class provides a case study for designing classes for user's convenience and for assuring data integrity. The class provides additional methods: `nextBeat` and `skipBeat` that either decrease or increase the note duration. These methods are used when a note is playing for the duration of several beats.

A **`Chord`** is a collection of `Note`s that are to be played at the same time. It can be initialized in a constructor to a given sequence of `Note`s, or `int`s (pitches) or `String`s (note names). It can also be modified later by adding notes to the chord. This class also includes the methods `nextBeat` and `skipBeat` that either decrease or increase the note duration for all notes in the `Chord`. But here we come with an interesting challenge. When the programmer decides that a given `Chord` should be played by adding it to the `TuneBucket`, she does not expect the `Chord` to change as the program plays the tune. So, we need to make a deep copy of the `Chord` before it starts playing, to assure that the mutation of the state as we progress through the beats has no ill effects. This provides a nice example for learning about the meaning of deep copy and illustrating the need for it.

The class **`Tune`** represents a chord that is to be played on the selected instrument. Each tune specifies the instrument number from the *MIDI* program and the `Chord` that should play. The class **`TuneBucket`** then contains 16 **`Tune`**s, one for each of the 16 instruments in the current *MIDI* program. The programmer can add to the `TuneBucket` one note, one `Chord`, a `Tune`, a collection of `Tunes`, and also clear the contents of the `TuneBucket` that stops playing all notes and removes all `Tunes` from the TuneBucket. The methods `nextBeat` and `skipBeat` work in the same way as for a single `Chord`.

To represent a melody, student starts with a sequence of notes. For example, the sequence of notes `(noteC,0,noteD,0,noteE,0,noteC,0)` represents the first phrase of the Frere Jacques tune. The pitch 0 represents a silent note, a pause. The next note to be played can be generated by a circular iterator, creating an infinite melody loop. We can combine two traversals over the melody with the appropriate delay in the second traversal to play a musical canon or we can play several instruments in parallel as an orchestra would do.

The *isworld* package: Making games

The *isdraw* library provides a `Canvas` for drawing simple shapes (rectangles, lines, circles, disks, and text) in any color and size. It defines an abstract class `World` that handles the creation of the frame with the `Canvas`, the drawing of the representation of the current world scene, the handling of the timer events, the key events, and mouse events. The programmer designs a class that extends the `World` and implements the methods that produce the changes in the `World` in response to the clock tick (`onTick` method), in response to a key press and release (`onKeyEvent`, `onKeyReleased`), and a method that draws the current state of the `World`. Optionally, the programmer can override the stubs of methods that respond to mouse events.

To start playing a tune at a given time, the `onTick` method includes a command that adds the selected `Tunes` to the `tickTunes TuneBucket`. The selected notes will play on the given instrument with duration measured in clock ticks. The `Tunes` added to the `keyTunes TuneBucket` play as long as the key is held down, regardless of the given duration. The `onKeyReleased` method does not affect the notes played --- it is provided so the programmer can take other actions when the key is released.

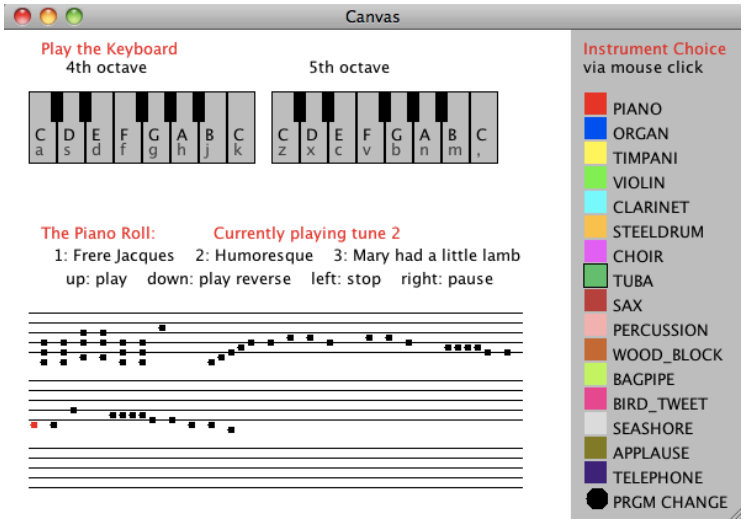
Testing support

The **SoundLib** library has been deliberately designed to support the test-first pedagogy. Every class in the *tunes* and *isworld* packages comes with methods that allow the programmer to check the effects or outcomes of methods defined in that class. The `MusicBox` class allows the programmer to check the current channel assignments to instruments (`getProgram(int channel)` method), and to check which tunes are currently playing provided by the `nowPlaying` method. The `Note` class includes a method `sameNote` that checks whether this note represents the same note as the given note. It verifies that the notes have matching pitch and duration (for example, the two notes represented by the `Strings "G4s2"` and `"A4f2"` are considered the same). The

`Tune` class and the `Chord` class both allow us to check their size, and whether they contain the given `Note`. The `TuneBucket` class allows us to check whether it contains a given note played on the given instrument, and it reports the size of the `TuneBucket`.

THE DEMO PROGRAM AND STUDENT PROJECTS

To illustrate some of the ways the library can be used, and to help the students to understand how to *program music* we have designed a demo program. The goal is to show the multiple ways of how the musical ideas can be represented, and how the user can observe and control the music that is played.

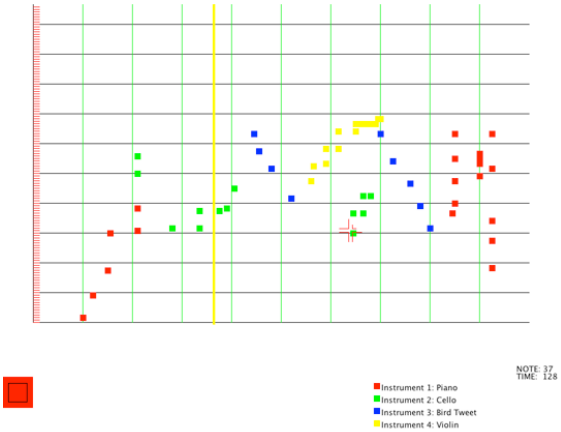


The right panel shows the current *MIDI program*: the assignment of instruments to the 16 MIDI channels. The user can choose any one of the instruments via mouse click. This way the students can hear how the different instruments sound. The displayed keyboard on the top left provides a guide to the user who wants to play a piano. The labels on the keys show the note that is played and the key press that plays the note. The note that is currently played is highlighted. The bottom left panel shows a piano roll. The musical staff shows the notes that will be played on each tick. The arrow keys stop, pause, reverse, and stop the playing of the selected tune. As the piano roll plays, the currently played notes are shown in red.

Student experiences and what they taught us

The first time we used the library we had no duration for the notes, no chords, and the instrument-note request had to be added to the `TuneBucket` one at a time. Yet students came up with truly engaging jazzy tunes, using silence between the notes to create the right timing. Before the second semester we improved the design with note duration and a more complex structures for creating musical compositions (almost everything described here except the mouse interactions and key press duration).
The final projects during the second semester were quite creative. In the frogger game a jazzy tune plays in the background, each collision and when the frog reached the other

bank of the river produces sound effects, and a great final tune plays when the game ends. A memory game asked the player to remember the sequence of square choices and music that went with it. One student sonified the Game of Life. Another project was a tool for composing music with playback (see below). The user could select one of four instruments, and pick the pitch to play at the given time on a graphical board that resembled the piano roll: the vertical dimension represented the pitch, the horizontal direction represented the timeline. Pressing the start key played the composition representing the current time by a thin horizontal line. This project made it clear that we need to add mouse events to the library, as all pitch selections were done by just moving the cursor using the arrow keys.



Another student tried to capture the melody played by playing the keys like a piano keyboard and playing it back on demand. His project motivated the re-design of the way the key press and release is handled in the current version of the library. Overall, the complexity and creativity exhibited in student projects confirmed the expected benefits of providing this library.

ACKNOWLEDGMENTS

The author would like to thank Erich Neuwirth for his inspiration and encouragement of exploration of programmatically generated music. His work on using music with spreadsheets and Logo have influenced the library design and its pedagogy.

REFERENCES

[1] http://www.ccs.neu.edu/javalib/SoundLib.
[2] Neuwirth, E., http://sunsite.univie.ac.at/musicfun/MidiCSD/.
[3] Proulx, V.K.., *Test-Driven Design for Introductory OO Programming*, SIGCSE Bulletin, 41(1), 2009.
[4] Sendova, E. (2001) *Modelling Creative Processes in Abstract Art and Music*, Eurologo 2001, Proceedings of the 8th European Logo Conference 21-25 August, Linz, Austria.