

Java Music Systems, Didkovsky

Score: JMSL's music notation package

JMSL's score package

JMSL has a programmable music notation package called `com.softsynth.jmsl.score`. The most important class in this package is a `Score`. The `ScoreFrame` class provides a GUI container and a menu bar for multiple `Scores`. You can provide your own GUI for `Score`'s various components. The new **portview** package allows `Score` gui components to be either Swing or AWT. Don't let the look of the `ScoreFrame` editing environment mislead you: a `Score` is an active object that can be created, launched, changed, self-modified, etc at runtime under program control.

The structure of a `Score` object is as follows:

- `Score` holds a reference to a `ScoreCollection`.
- `ScoreCollection` is a `SequentialCollection` of `Measure`.
- `Measure` is a `ParallelCollection` of `Staff`
- `Staff` is a `ParallelCollection` of `Track`
- `Track` is a `MusicList` of `Note`
- `Note` implements `InstrumentPlayable`, and contains graphic as well as musical data and methods.

`Score` allows the user to **enter and edit notes with the mouse**, but its real strength lies in its **programmability**, which breaks down into three areas:

1. Adding algorithmically generated notes to a score (where note durations are specified)
2. Transcribing arbitrarily generated musical events
3. Transforming selected notes in a score

I Adding measures to a score

Here we demonstrate:

- How to create a new score.
- How to add it to a frame for display and editing.
- How to use JMSL's api for adding measures with user defined time signatures.

A new `Score` is created like so:

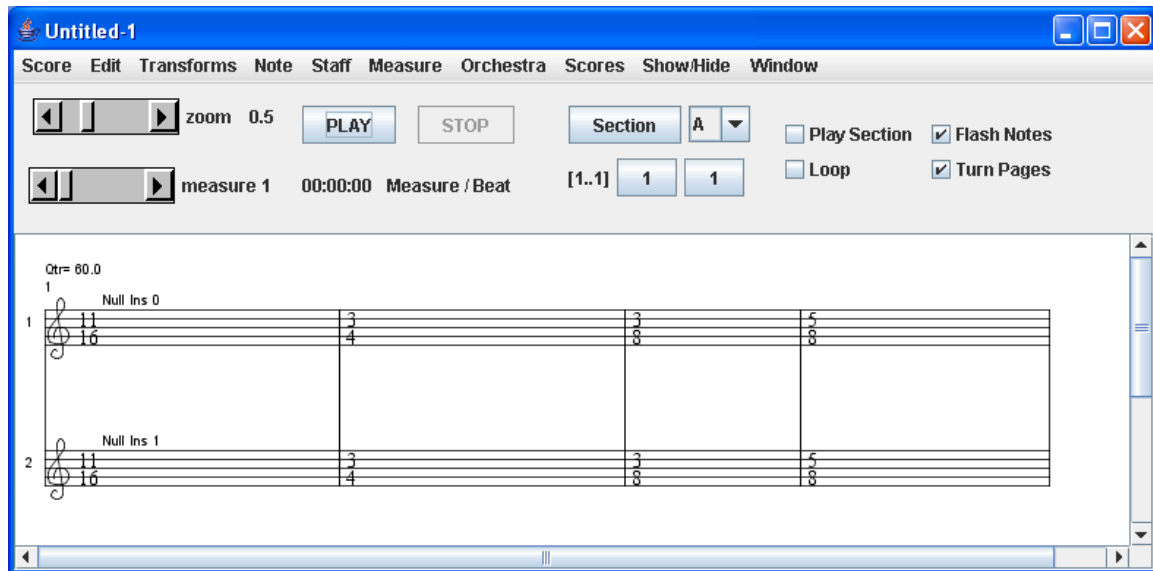
```
// 2 staves, width, height
Score score = new Score(2, 800, 400);
```

A score can be added to a ScoreFrame for display and editing. ScoreFrame has a powerful set of menus for score editing. Adding a score to a ScoreFrame is done like this:

```
ScoreFrame scoreFrame = new ScoreFrame();
scoreFrame.addScore(score);
```

A blank measure with an arbitrary time signature can be added to a score with the addMeasure() method. For example:

```
score.addMeasure(11, 16);
score.addMeasure(4, 4);
// add a measure with random time signature
score.addMeasure(JMSLRandom.choose(3, 7),
    (int)Math.pow(2, JMSLRandom.choose(2, 4)));
```



The complete source for an applet that opens the ScoreFrame shown above does is shown below:

```
/**
 * Add measures with various time signatures to a JMSL score
 *
 * @author Phil Burk and Nick Didkovsky
 */
/*
 * (C) 1997 Phil Burk and Nick Didkovsky, All Rights Reserved
 * JMSL is based upon HMSL (C) Phil Burk, Larry Polansky and David Rosenboom.
 */

package jmsltutorial;

import java.awt.*;
import com.softsynth.jmsl.*;
import com.softsynth.jmsl.score.*;

public class JScoreToot01 extends java.applet.Applet {

    Score score;
    ScoreFrame scoreFrame;
```

```

public void init() {
    // setIsApplet set to true to disable menu items that would
    // cause security exceptions when running as an applet
    JMSL.setIsApplet(true);
    JMSLRandom.randomize();
    setBackground(Color.yellow);
    add(new Label("Your JMSL score will open in a new window"));
}

/* When applet starts up, display the scoreframe */
public void start() {
    synchronized (JMSL.class) {
        JMSL.clock = new DefaultMusicClock();
        JMSL.clock.setAdvance(0.1);
        JMSL.scheduler = new EventScheduler();
        JMSL.scheduler.start();

        // 2 staves, width, height
        score = new Score(2, 800, 400);
        scoreFrame = new ScoreFrame();
        scoreFrame.addScore(score);

        score.addMeasure(11, 16);
        score.addMeasure(3, 4);
        score.addMeasure(3, 8);
        score.addMeasure(5, 8);
        // add three measures with random time sigs
        score.addMeasure(JMSLRandom.choose(3, 7), (int) Math.pow(2,
JMSLRandom.choose(2, 4)));
        score.addMeasure(JMSLRandom.choose(3, 7), (int) Math.pow(2,
JMSLRandom.choose(2, 4)));
        score.addMeasure(JMSLRandom.choose(3, 7), (int) Math.pow(2,
JMSLRandom.choose(2, 4)));
        scoreFrame.setVisible(true);
        scoreFrame.setSize(900, 600);
        score.render();
    }
}

/* When applet stops, hide the scoreframe */
public void stop() {
    synchronized (JMSL.class) {
        if (scoreFrame != null) {
            scoreFrame.setVisible(false);
            scoreFrame.dispose();
            Score.deleteCanvas();
            SelectionBuffer.disposeEditFrame();
            scoreFrame = null;
            score = null;
        }
        JMSL.scheduler.stop();
        JMSL.closeMusicDevices();
    }
}
}

```

II Adding algorithmically generated notes to a Score with addNote()

Score provides a method for adding Notes, called `addNote()`. Every call to `addNote()` appends a new Note onto the end of the current staff of the current measure. Measures are added as needed.

The method signature of `addNote()` is as follows:

```
public Note addNote(double dur, double pitch, double amp, double hold)
```

- **dur**: JMSL's score package will search for the duration closest to `dur`, and build a Note object from it. It will then insert the Note into the current Staff of the current Measure. Supported durations include core durations: whole, half, quarter, eighth, sixteenth, 32nd, 64th, and 128th, where quarter = 1.0. Also supported are the dotted versions of these core durations, and 3, 5, 7, 11 tuplet versions of these core durations.
- **pitch**: a double 0..127 (midi style), where C6 (middle C) is 60, and 0 implies a rest.
- **amp**: a double 0..1 where 0 is silent and 1.0 is max amplitude
- **hold**: the sustain duration for this note

For example:

```
score.addNote(1.0, NoteFactory.MIDDLE_C, 0.5, 0.8);           // add C qtr note
score.addNote(0.5, NoteFactory.MIDDLE_C+1, 0.5, 0.4);       // add C# 8th note
score.addNote(0.33333, NoteFactory.MIDDLE_C+2, 0.5, 0.2);   // add D 8th note triplet
score.addNote(0.33333, 0, 0, 0.2);                          // add 8th note triplet rest
```

Setting the insertion point of added notes

Measures are numbered 0..(numMeasures-1). Staves are numbered 0..(numStaves-1). So once you've filled up staff 0 with notes, you can rewind to the beginning of the score and add notes to staff 1, like so:

```
score.rewind();
score.setCurrentStaffNumber(1);
score.addNote(...);
```

Notice that you can add as many notes as you like, and measures will automatically get added as needed (inheriting the time signature from the last measure). You may also jump to any measure and add notes into it with a call to `score.setCurrentMeasureNumber(int n)`; However it is your responsibility that this measure exists (call `score.size()` to see how many measures are there).

Manipulating the properties of an added Note: beaming and chords

The `addNote()` method returns the new Note it created and added to the Score. You can get a hold of this note and manipulate it. You may set its beaming, for example, or add a mark.

```
Note note = score.addNote(1.0, 60, 0.5, 0.8);
note.setBeamedOut(true); // beam to next note
note.setMark(Note.MARK_ACCENT);
```

You may also build a chord with a call to:

```
Note note = score.addNote(1.0, 60, 0.5, 0.8);
note.addInterval(67);
```

The the JMSL docs for Note for other methods applicable to note.

III Adding algorithmically generated notes to a Score with the Transcriber

The previous section describes the `addNote()` method which adds a `Note` with a standard duration to a score. However, consider the problem of notating event durations which do not conform to traditional durations. The user may have any number of musical events scattered arbitrarily over time, generated stochastically for example, and wishes to create a score of this material in common music notation. JMSL's `Transcriber` class analyzes such musical material, and loads a `Score` with a customizable transcription.

Input to the Transcriber

The input to JMSL's `Transcriber` is a `MusicShape` of musical events, a `Vector` of `TimeSignatures` and `Tempos` which provide a template for the transcription, and a list of permitted beat divisions (ie ways of subdividing a beat: triplet, quintuplet, septuplet, etc). The `MusicShape`'s first four dimensions must be duration, pitch, amplitude, hold. Higher dimensions will be preserved by the transcription process. **IMPORTANT: the 0th dimension (duration) must first be integrated to convert durations to absolute timestamps. This is done with `myMusicShape.integrate(0)`. If you cannot afford to lose the duration info held in dimension 0, first `clone()` the `MusicShape` and integrate the clone.**

Specifying which beat subdivisions the transcriber may use

For each `Measure` specified in the `Vector` of `TimeSignatures`, the transcriber's goal is to find a quantized path of minimum error through the musical events using the permitted beat divisions (ie binary sixteenth notes, triplets, quintuplets, etc). The user can specify which beat subdivisions the transcriber may consider. Each subdivision is described by the `BeatDivisionScheme` class. All `BeatDivisionSchemes` under consideration for a particular piece are held in the `BeatDivisionSchemeList`. By adding `BeatDivisionSchemes` to the master list, the user can customize the transcriber by including, for example, only eighth note triplets, and quarter note septuplets. Additional customization can be achieved by specifying the minimum number of elements to be present for a `BeatDivisionScheme` to be considered. For example, one might specify that the minimum number of notes in an eighth note triplet be three, eliminating the possibility of a triplet containing two notes: one with twice the duration of the other, for example. One can massage this threshold and in general, require a minimum of any value between 1 and n notes in an n-tuplet. For all this customization however, a default setting is available simply by calling `BeatDivisionSchemeList.defaultSetup()` ;

Simple Transcription Example

Here we add 17 algorithmically generated elements to a `MusicShape` and transcribe them.

* First we set up a default beat division scheme list:

```
BeatDivisionSchemeList.defaultSetup();
```

* Now we add elements with random durations to a `MusicShape`

```
MusicShape melody1 = new MusicShape(4);
for (int i = 0; i < 17; i++) {
    double duration = JMSLRandom.choose(3.0);
    double pitch = JMSLRandom.choose(60, 84);
    double amp = JMSLRandom.choose(0.2, 0.7);
    double hold = duration * 0.8;
    melody1.add(duration, pitch, amp, hold);
}
```

```
}
```

* Now we need to convert the durations to absolute time stamps:

```
melody1.integrate(0);
```

* Here we create a Score and a Transcriber

```
Score score = new Score(2, 1024, 800);  
score.addMeasure();
```

```
Transcriber transcriber = new Transcriber();  
transcriber.setScore(score);
```

* Ready to transcribe! Notice that we set the insertion point to measure 0 with `rewind()`, and set the staff number to staff 0 (top staff).

```
score.setCurrentStaffNumber(0);  
score.rewind();  
transcriber.setSourceMusicShape(melody1);  
transcriber.transcribe();
```

The above four lines need to be in a `try{} catch{} block` which we do not show here for brevity, but you should know that `ElementMissedException` and `SearchPathListExpansionException` exceptions may be thrown.

* Finally, display the Score in a ScoreFrame:

```
ScoreFrame myScoreFrame = new ScoreFrame();  
myScoreFrame.addScore(score);  
myScoreFrame.setVisible(true);
```

The example above was taken from `jmsltestsuite.TestTranscribeSimple` whose source ships with JMSL

Specifying Time Signatures and Tempos

The above example used a time signature of 4/4 time throughout, and a tempo of 60 bpm. That means if an element's duration was 1.0, then it was interpreted as 1 quarter note (since a quarter note last 1.0 seconds at 60 bpm).

Before transcribing, you can provide a template for measures and tempi by providing the Transcriber with a Vector of `TempoTimeSignatureHolder` objects. The example below adds three such objects to the scheme, starting with a measure of 2/4 at 60bpm, followed by one measure of 2/4 at 120bpm, followed by a measure of 2/4 at 60bpm again. The transcriber will scale the timestamps of the input `MusicShape` to flow into the scheme provided below.

```
Vector tsVector = new Vector();  
tsVector.addElement(new TempoTimeSignatureHolder(new Tempo(60), new  
TimeSignature(2, 4)));  
// now double tempo!!!  
tsVector.addElement(new TempoTimeSignatureHolder(new Tempo(120), new  
TimeSignature(2, 4)));  
tsVector.addElement(new TempoTimeSignatureHolder(new Tempo(60), new  
TimeSignature(2, 4)));  
transcriber.setTempoTimeSignatures(tsVector);
```

The default behavior of the Transcriber is to keep adding measures with the last specified

tempo and time signature to the score as more elements are transcribed. However you can toggle this behavior with `myTranscriber.setLoopTimeSignatures(true)` which will cycle through this template as measures are added (ie using the example above, the 4th measure would be 2/4 at 60 bpm, the fifth measure would be 2/4 at 120 bpm, etc)

The example above was taken from `jmsltestsuite.TestTranscribe3` whose source ships with JMSL

Transcriber Listener provides call-backs during the Transcription process

One of the strengths of having a programmable transcriber in an algorithmic music API is that you can preserve the intelligence that your system has already assigned to the music it has generated, as opposed to exporting your music to a MIDI file and losing all of it. Through the `TranscriberListener` interface, JMSL lets you massage Notes as the Transcriber adds them to a Score. For example, you might assign algorithmically generated lyrics to Notes, or algorithmically generated marks (accents, staccato, etc). The `TranscriberListener` interface provides you with this capability. The interface defines two methods shown below:

```
/** Called whenever a Note is added to the Score by the Transcriber. Do
what you want to the Note */
public void noteAdded(Score score, Note note);

/**
Notify listener of note events that will spill over into the next
measure, ie notes that are found within the current time window but
will round up to beat 1 of the following measure. Note that musicShape
could be null if there was no spillover!
*/
public void notifyCarriedOverMusicShape(Score score, int
currentMeasureNumber, MusicShape musicShape)
```

If we wanted to randomly assign an accent to a Note as it is being transcribed we could define a class called `MyNoteAccenter` that implements `TranscriberListener` and defines `noteAdded` as follows:

```
public void noteAdded(Score score, Note note) {
    if (note.isRest()) {
        return;
    }
    if (JMSLRandom.choose() < 0.3) {
        note.setMark(Note.MARK_ACCENT);
    }
}

public void notifyCarriedOverMusicShape(Score score, int
currentMeasureNumber, MusicShape musicShape) {} // do nothing
```

Then, before calling `transcribe()`, hand an instance of this class to the transcriber:
`transcriber.addTranscriberListener(new MyNoteAccenter());`

The result will be a transcribed Score some of whose Notes will show accent marks.

The example above was taken from `jmsltestsuite.TestTranscribe8` whose source ships with JMSL

IV Transforming notes algorithmically

JScore supports three kinds of transforms which operate on Notes already in a Score:

1. **Unary Copy Buffer Transform** - operates on notes contained in the Copy Buffer.
Example: result = retrograde of copy buffer.
2. **Binary Copy Buffer Transform** - operates on two buffers of notes contained in Auxilliary Buffers 1 and 2
Example: result = pitch-averaged notes from aux1 and aux2
3. **Note Properties Transform** – operates on selected Notes “in place”
Examples: apply accent mark to each Note, or randomly displace pitch of selected Notes

Plug-ins

You may add your transforms to the ScoreFrame menu explicitly (see “Adding your custom transform to ScoreFrame’s menu” below), or you may move the compiled .class files into a folder called “jmsl_plugins”, which must be in your classpath. Upon startup, ScoreFrame scans the classpath and if it finds a directory called “jmsl_plugins”, it will look for these transforms and build menus for them automatically. It also scans for Instruments, SynthNotes, and ScoreOperators.

When you copy your plug-ins into jmsl_plugins, your directory structure must preserve the package structure. So if you have a transform called InvertTransform which is in the package com.didkovsky, then inside jmsl_plugins you need a directory called “com”, containing a directory called “didkovsky” which in turn contains the class file “InvertTransform.class”

Unary Copy Buffer Transform example: Retrograde Transform

The Retrograde transform is a Unary Copy Buffer Transform which reverses the order of the notes in the copy buffer. This simply moves Notes around in the CopyBuffer. It does not change pitch or duration.

You can design your own UnaryCopyBufferTransform by extending UnaryCopyBufferTransform and overriding

```
public void operate(CopyBuffer copyBuffer)
```

The operate() method receives the CopyBuffer, which contains the Notes upon which to operate. CopyBuffer is a java.util.Vector, so Notes in the buffer can be examined, added, deleted, and mutated like any Object in a Vector.

Let us examine the source code for the Retrograde Transform, which is below. Notice the constructor assigns a unique name to the transform. This name will show up in the ScoreFrame's menu when it is added later.

```
package com.softsynth.jmsl.score.transforms;

import com.softsynth.jmsl.score.CopyBuffer;
import com.softsynth.jmsl.score.UnaryCopyBufferTransform;

public class RetrogradeTransform extends UnaryCopyBufferTransform {
    public RetrogradeTransform() {
        setName("Retrograde");
    }

    /** Implement this method to do whatever you want to CopyBuffer arg. */
    public void operate(CopyBuffer copyBuffer) {
        int start =0;
        int end = copyBuffer.size()-1;
        while (start < end) {
            Object temp = copyBuffer.elementAt(start);
            copyBuffer.setElementAt(copyBuffer.elementAt(end), start);
```

```

        copyBuffer.setElementAt(temp, end);
        start++;
        end--;
    }
}

```

Adding your UnaryCopyBufferTransform to ScoreFrame's menu

You may add a custom Transform to a ScoreFrame one of two ways:

1) Programmatically, with a call to

```
public void addUnaryCopyBufferTransform(UnaryCopyBufferTransform transform).
```

Consult the documentation for other versions of this method which allow you to add to your own submenus, and add shortcut keys.

Example:

```
scoreFrame.addUnaryCopyBufferTransform(new MyWeirdTransform());
```

2) By putting your transform's .class file in the jmsl_plugins folder

Operations on Note:

When you implement your own transform, you override `operate()`, as discussed above. `Operate()` is passed the contents of the current copy buffer. The CopyBuffer is simply a Vector of Note objects. You can get and set a Note's pitch and you can get and set a Note's duration. After changing one or the other (or both), you must call `NoteFactory.updateFromPitch(note)` or `NoteFactory.updateFromDur(note)`. Other operations are available as well, but for now let's stick to pitch and duration.

For Example:

```

public void operate(CopyBuffer copyBuffer) {
    for (Enumeration e=copyBuffer.elements(); e.hasMoreElements(); ) {
        Note note = (Note)e.nextElement();

        // transpose up by a fifth
        note.setPitchData(note.getPitchData() + 7);
        NoteFactory.updateFromPitch(note);

        // double the duration
        note.setDurationData(note.getDurationData() * 2);
        NoteFactory.updateFromDur(note);
    }
}

```

BinaryCopyBufferTransform example: Morphological Weighted Mean Transform

A Binary Copy Buffer Transform operates on two copy buffers, called Aux Buffer 1 and Aux Buffer 2. Its `operate()` method loads the normal Copy Buffer with its results, from where it can be pasted back into the score.

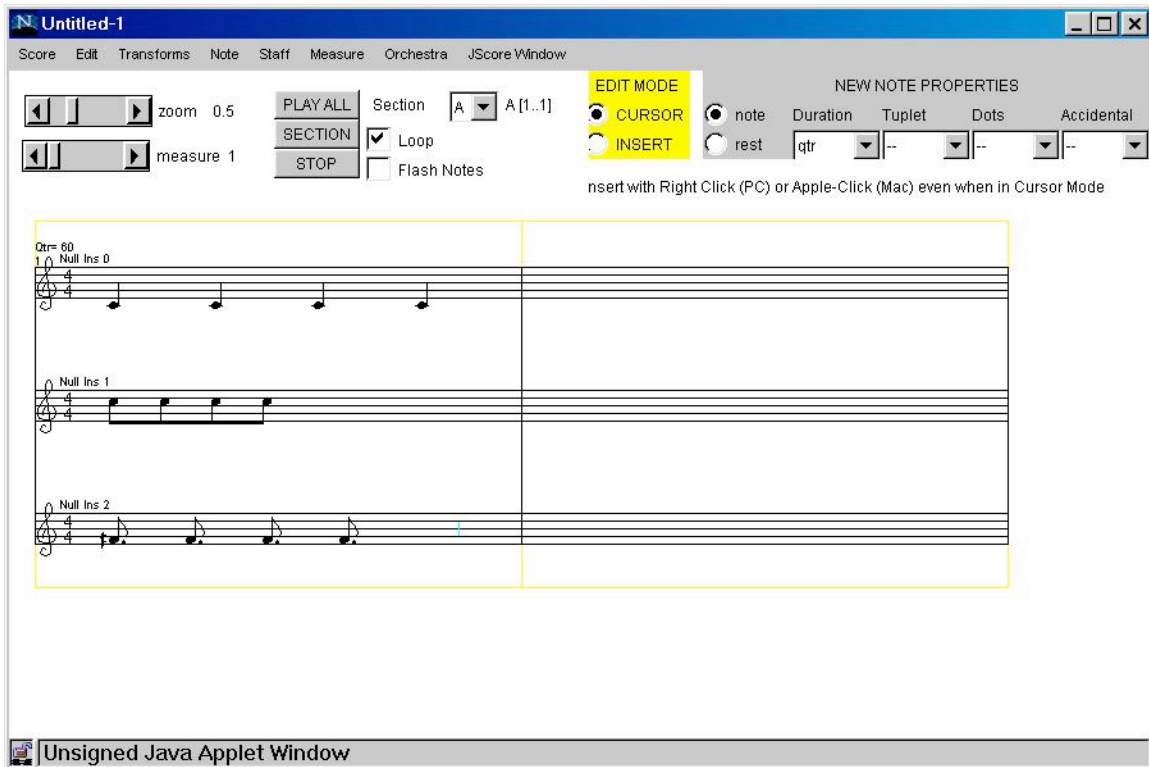
We will implement a Binary Copy Buffer Transform that generates a weighted morphological

mean of the two aux sources. We implement a mutation function where each resulting $Note_n$ is the function of $Aux1Note_n$ and $Aux2Note_n$

Specifically:

- the pitch of the resulting $Note_n$ = mean pitch of $Aux1Note_n$ and $Aux2Note_n$
- the duration of the resulting $Note_n$ = mean duration of $Aux1Note_n$ and $Aux2Note_n$

The score below shows the result of a mutation mean transform. Staff 1 and 2 are the sources, while staff 3 is the result. Notice that the average duration of quarter note and an eighth note results in a dotted eighth, and that the average of middle C and the C above is the tritone F#.



The screenshot shows a music notation software window titled "Untitled-1". The interface includes a menu bar (Score, Edit, Transforms, Note, Staff, Measure, Orchestra, JScore Window) and a toolbar with zoom (0.5) and measure (1) controls. A "NEW NOTE PROPERTIES" panel is visible, showing options for "EDIT MODE" (CURSOR, INSERT), "note" or "rest", "Duration" (qtr), "Tuplet", "Dots", and "Accidental". The score area displays three staves: "Null Ins 0", "Null Ins 1", and "Null Ins 2". The first two staves show source material, and the third staff shows the result of a mutation mean transform, including a dotted eighth note and a tritone F#.

Here we show the source for the mutation mean transform:

```
public class MutationMeanTransform extends BinaryCopyBufferTransform {  
    public MutationMeanTransform() {  
        setName("MutationMean");  
    }  
  
    public void operate(CopyBuffer fromBuffer1, CopyBuffer fromBuffer2, CopyBuffer toBuffer)  
    {  
        toBuffer.removeAllElements();  
        Enumeration e1=fromBuffer1.elements();  
  
        // load all of aux1 into result buffer  
        while (e1.hasMoreElements()) {  
            toBuffer.addElement(e1.nextElement());  
        }  
    }  
}
```

```

// now enumerate through again and calculate new pitch and dur,
// stop when either buffer has no more elements
Enumeration e2=toBuffer.elements();
Enumeration e3=fromBuffer2.elements();

while (e2.hasMoreElements() && e3.hasMoreElements()) {
    Note n1 = (Note)e2.nextElement();
    Note n2 = (Note)e3.nextElement();
    n1.setPitchData((int)(n1.getPitchData() + n2.getPitchData() /2));
    NoteFactory.updateFromPitch(n1);
    n1.setDurationData((n1.getDurationData() + n2.getDurationData() /2));
    NoteFactory.updateFromDur(n1);
}
}
}

```

Adding your BinaryCopyBufferTransform to ScoreFrame's menu

You may add a custom Transform to a ScoreFrame one of two ways:

1) Programmatically, with a call to

```
public void addBinaryCopyBufferTransform(BinaryCopyBufferTransform transform).
```

Consult the documentation for other versions of this method which allow you to add to your own submenus, and add shortcut keys.

Example:

```
scoreFrame.addBinaryCopyBufferTransform(new MyWeirdTransform());
```

2) By putting your transform's .class file in the jmsl_plugins folder

NotePropertiesTransform example: Serial Pitch Transform

A NotePropertiesTransform operates on the selected Notes in place (ie you do not have to copy them first). These Notes are in a Selectionbuffer. Below is the source for SerialTransform which runs through the selected notes and resets their pitch data according to a 12 tone row.

```

package com.punosmusic.transforms;

import com.softsynth.jmsl.score.*;
import com.softsynth.jmsl.*;
import java.util.*;

/**
 * For JScore. Just blort out some 12 tone row pitches and keep rhythms
 the same
 *
 * @author Nick Didkovsky, copyright 2000 Nick Didkovsky, all rights
 reserved
 */
public class SerialTransform extends NotePropertiesTransform {

    MusicShape row;

```

```

public SerialTransform() {
    setName("Serial Transform");
    row = new MusicShape(1);
    for (int i = 0; i < 12; i++) {
        row.add(60 + i);
    }
}

public void operate(Score score, SelectionBuffer selectionBuffer) {
    row.scramble(0, row.size() - 1, 0);
    int kount = 0;
    for (Enumeration e = selectionBuffer.elements();
e.hasMoreElements();) {
        Note note = (Note) e.nextElement();
        if (!note.isRest()) {
            double pitch = row.get(kount % row.size(), 0);
            kount++;
            pitch += 12 * JMSLRandom.choose(3);
            note.setPitchData(pitch);
            // Use NoteFactory to recalculate staff level,
accidental, and
            // stem direction
            NoteFactory.setLevelPitch(note, note.getPitchData());
            if (note.isChord() || note.isInterval()) {
                Note.resortChord(note);
            }
        }
    }
}

public static final String copyright = "copyright (C) 2000 Nick
Didkovsky, all rights reserved";
}

```

Adding your NotePropertiesTransform to ScoreFrame's menu

You may add a custom Transform to a ScoreFrame one of two ways:

1) Programmatically, with a call to

```
public void addNotePropertiesTransform (NotePropertiesTransform transform).
```

Consult the documentation for other versions of this method which allow you to add to your own submenus, and add shortcut keys.

2) By putting your transform's .class file in the jmsl_plugins folder

ScoreOperators perform an arbitrary action on a Score and return a String result

The ScoreOperator interface allows the programmer to define arbitrary operations on a Score. The ScoreOperator should return a String in its getResultString() method to provide a report of

activity (such as a total of durations or simply a message saying “done). When ScoreFrame executes a ScoreOperator, it will open a Frame with a TextArea displaying the String returned by getResultString().

Below is an example of a ScoreOperator that totals the durations in a Score’s selection buffer. This is handy as it lets the user select a range of notes and see what their total is (good when rhythms get confusing).

```
public class TotalSelectedDurations implements ScoreOperator {

    double totalDur = 0;
    int totalNotes = 0;

    public String getName() {
        return "Total Selected Durations";
    }

    public void operate(Score score) {
        SelectionBuffer buf = score.getSelectionBuffer();
        totalDur = 0;
        totalNotes = 0;
        for (Enumeration e = buf.elements(); e.hasMoreElements();) {
            Note note = (Note) e.nextElement();
            totalDur += note.getDurationData();
            totalNotes++;
        }
    }

    public String getResultString() {
        return "Total notes: " + totalNotes + ", total duration: " +
            totalDur;
    }
}
```

IMPORTANT: Since ScoreOperator gets a handle to the Score, you can do anything you like to the score, including adding measures, adding Notes, altering material, analyzing it, etc etc etc. It is not just a passive observer of the Score, it can mutate it.

Adding your ScoreOperators to ScoreFrame's menu

You may add a custom ScoreOperator to a ScoreFrame one of two ways:

1) Programmatically, with a call to

```
public void addScoreOperator (ScoreOperator scoreOperator).
```

Consult the documentation for other versions of this method which allow you to add to your own submenus, and add shortcut keys.

2) By putting your ScoreOperator’s .class file in the jmsl_plugins folder

V Signal Processing: Patching a JMSL Score Orchestra

You can route signal sources to Signal Processing Instruments in JMSL Score. Signal processors affect the output of the JSyn instrument.

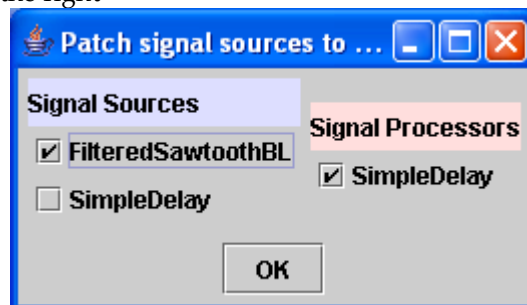
You can also add PlayLurkers to selected staves. PlayLurkers receive notification of Notes being played on other staves.

SIGNAL PROCESSING

SignalProcessingInstruments are created when a JSyn SynthNote is found in the jmsl_plugins classpath, and is found to contain a SynthInput called "input".

For this example, we will use classes already included in JMSL and JSyn in case you do not have any plugins that fit the bill.

- 1) Start ScoreFrame and create new Orchestra (Orchestra -> Orchestra -> New Orchestra)
- 2) Import a FilteredSawtoothBL SynthNote for the top staff (Orchestra -> JSyn -> Import JSyn SynthNote), and select OK on the Dialog that opens for com.softsynth.jsyn.circuits.FilteredSawtoothBL
- 3) Import a SignalProcessing SynthNote for the second staff (Orchestra -> JSyn -> Import JSyn SignalProcessor), and click OK on the Dialog that opens for com.softsynth.jmsl.jsyn.circuits.SimpleDelay
- 4) Add notes to the top staff
- 5) Add notes to the bottom staff
- 6) Select play section and loop checkboxes, and play
- 7) Patch the FilteredSawtoothBL to the Simple Delay: Select Orchestra -> Patch Bay -> Patch Orchestra and check the signal source on the left to route it to the signal processor on the right



- 8) Click OK and you will hear delay on the melody
- 9) Double click on any note in the SimpleDelay staff and change its feedback and delay length parameters.

PLAY LURKERS

A PlayLurker can be added to any MusicShape and receives notification of the element played. PlayLurker is an inter.\ace. You implement public void notifyPlayLurker(double playTime, MusicJob list, int index) to handle the incoming data.

The source below shows a PlayLurking instrument that receives notification of another track playing. Depending on a probability threshold, it will simultaneously play its own instrument some transposition interval away from the incoming pitch. The result is a long sustained ghostly pitch being occasionally performed along with a source melody. This class was used in Nick Didkovsky's Sabbath Bride

(<http://www.punosmusic.com/pages/sabbathbride/sabbathbride.html>)

```
package com.punosmusic.sabbathbride;
import com.softsynth.jmsl.*;
import com.softsynth.jsyn.*;
import com.softsynth.jmsl.jsyn.JSynInsFromClassName;
import com.softsynth.jmsl.score.*;
import java.io.*;

/** Double a melody note with increasing probability
 * @author Nick Didkovsky, April 22, 2003 */
public class HeavyBonesSine extends JSynInsFromClassName implements PlayLurker
{

    double[] data = new double[4];
    double pStep = 0.98 / (39 * 10);
    double pPlay = 0.0;

    public HeavyBonesSine() {
        setMaxVoices(8);
        setSynthNoteClassName("patches.SlowSine");
        System.out.println("nicer");
        //super(8, "patches.SlowSine");
    }

    public void notifyPlayLurker(double playTime, MusicJob list, int index) {
        Track track = (Track)list;
        Score score = track.getScore();
        Note note = (Note)track.get(index);
        pPlay += pStep;
        // System.out.println("p(Play)=" + pPlay);
        if (!note.isRest()) {
            if (JMSLRandom.choose() < pPlay) {
                double pitch = note.getPitchData();
                double duration = note.getDurationData();
                data[0] = JMSLRandom.choose(5.0, 10.0);
                int transpose = JMSLRandom.choose(5) - 2;
                data[1] = pitch + 12 * transpose ;
                data[2] = JMSLRandom.choose(0.5, 1.0);
                data[3] = 0.95 * data[0];
                this.play(playTime, 1.0, data);
            }
        }
    }

    public static void main (String args[]) {
        try {
            Synth.startEngine(0);
            HeavyBonesSine ins = new HeavyBonesSine();
            java.io.PrintWriter pout = new java.io.PrintWriter(new
java.io.FileOutputStream("heavybonessine.xml"));
            (new com.softsynth.jmsl.util.SimpleXMLSaver(ins,
"jmslscoreinstrument")).writeXML(pout);
            pout.close();
            Synth.stopEngine();
            System.exit(0);
        }
        catch (IOException e) {
            System.out.println("Whoops " + e);
        }
    }
}
```

```
}  
}
```

VII Create your custom JScore environment

It's important to repeat here that your transforms can be added to `jmsl_plugins` folder. If this folder is in your classpath, `ScoreFrame` will scan for any transforms, `Instruments`, `SynthNotes`, or `ScoreOperators` there and automatically build menus for them.

However, you can alternatively create your own custom `ScoreFrame` by directly subclassing, and add your own transforms, etc. Here is the source of an example `ScoreFrame`. Of course you would not expect this exact code to compile since you do not have the classes that are shown in bold below, but you can use this as a framework.

Why do this instead of using plug-ins? At least two reasons:

- 1) currently `ScoreFrame` does not scan for plug-ins when running as an applet.
- 2) You can also do other things to the scoreframe gui by subclassing, such as add your own menu by accessing `myScoreFrame.getMainMenuBar()`, or adding a panel using `myScoreFrame.add(BorderLayout.SOUTH, mySpecialPanel)`;

```
package didkovsky2608;

import java.awt.event.*;

import com.didkovsky.portview.swing.ViewFactorySwing;
import com.softsynth.jmsl.JMSL;
import com.softsynth.jmsl.midi.MidiIO_JavaSound;
import com.softsynth.jmsl.score.*;

/**
 * Nick's custom ScoreFrame
 *
 * @author Nick Didkovsky, copyright 2000 Nick Didkovsky, all right reserved
 */

public class HyperScoreFrame extends ScoreFrame {

    public HyperScoreFrame() {
        super();
        addUnaryCopyBufferTransform(new icmc2000workshop.ConvergeToMeanTransform());
        addUnaryCopyBufferTransform(new jmsltutorial.HocketTransform());
        addBinaryCopyBufferTransform(new jmsltutorial.MutationMeanTransform());
    }

    public static void main(String args[]) {
        JMSL.setViewFactory(new ViewFactorySwing());
        JMSL.clock.setAdvance(0.1);
        JMSL.midi = MidiIO_JavaSound.instance();

        final HyperScoreFrame hyperScoreFrame = new HyperScoreFrame();
        Score score = new Score(2, 1024, 600);
        score.addMeasure();
        hyperScoreFrame.addScore(score);
        hyperScoreFrame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                if (JMSL.midi != null)
                    JMSL.midi.closeDevices();
                hyperScoreFrame.savePrefs();
                System.exit(0);
            }
        });

        hyperScoreFrame.loadPrefs();
        hyperScoreFrame.setVisible(true);
    }
}
```

VIII The portview package supports GUI design which is switchable between AWT and Swing

Components

New to JMSL v103 is the `com.didkovsky.portview` package. If you build your GUI with components purely created by the portview ViewFactory, then your GUI's can be switched between AWT to Swing. All Score components and control panels are built using this package so it is worthwhile learning more about it [here](#).

View Factory – awt/Swing portable gui components

A ViewFactory is an interface that defines various Component-creating methods such as:

```
public PVPanel createPanel();
public PVButton createButton(String string);
public PVLabel createLabel(String string);
```

You can register a ViewFactory with JMSL by calling one of the following:

```
JMSL.setViewFactory(new ViewFactorySwing());
JMSL.setViewFactory(new ViewFactoryAWT());
```

The ViewFactory is responsible for generating Frames, Checkboxes, Labels, Panels, Buttons, MenuItems, Menus, etc at runtime.

Here is an example of building a gui in pure AWT:

```
/*
 * Created by Nick on Nov 14, 2004
 *
 */
package didkovsky2608;

import java.awt.*;
import java.awt.event.*;

/**
 * @author Nick Didkovsky, (c) 2004 All rights reserved, Email:
 *         didkovn@mail.rockefeller.edu
 */
public class SimpleFrameAWT extends Frame implements ActionListener {

    Button myButton;
    Label myLabel;

    public SimpleFrameAWT() {
        super("AWT Frame");
        setLayout(new BorderLayout());
        add(BorderLayout.WEST, myButton = new Button("CLICK ME FOR
DATE"));
        add(BorderLayout.EAST, myLabel = new Label("WATCH ME FOR THE
DATE"));
        myButton.addActionListener(this);
    }

    public void actionPerformed(ActionEvent ev) {
        Object source = ev.getSource();
        if (source == myButton) {
            String theDate = new java.util.Date().toString();

```

```

        myLabel.setText(theDate);
    }
}

public static void main(String[] args) {
    SimpleFrameAWT f = new SimpleFrameAWT();
    f.pack();
    f.setVisible(true);
    f.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
}
}

```

Here is the same class, rewritten with portview support. Note the use of `getComponent()` which returns the underlying Swing or AWT Component, which must be retrieved in order to add to a layout. Also note that `jmsl.view` provides a few adapters for some PVxxx interfaces such as `PVFrameAdapter` and `PVLabelAdapter`. You can use these or call `factory.createxxx()`.

```

/*
 * Created by Nick on Nov 14, 2004
 *
 */
package didkovsky2608;

import java.awt.BorderLayout;
import java.awt.event.*;

import com.didkovsky.portview.*;
import com.didkovsky.portview.swing.ViewFactorySwing;
import com.softsynth.jmsl.JMSL;
import com.softsynth.jmsl.view.PVFrameAdapter;

/**
 * @author Nick Didkovsky, (c) 2004 All rights reserved, Email:
 *         didkovn@mail.rockefeller.edu
 */
public class SimpleFramePortView extends PVFrameAdapter implements
ActionListener {

    PVButton myButton;
    PVLabel myLabel;

    public SimpleFramePortView() {
        super("PortView frame");
        ViewFactory factory = JMSL.getViewFactory();
        setFrameLayout(new BorderLayout());
        add(BorderLayout.WEST, (myButton = factory.createButton("CLICK
ME FOR DATE")).getComponent());
        add(BorderLayout.EAST, (myLabel = factory.createLabel("WATCH ME
FOR THE DATE")).getComponent());
    }
}

```

```

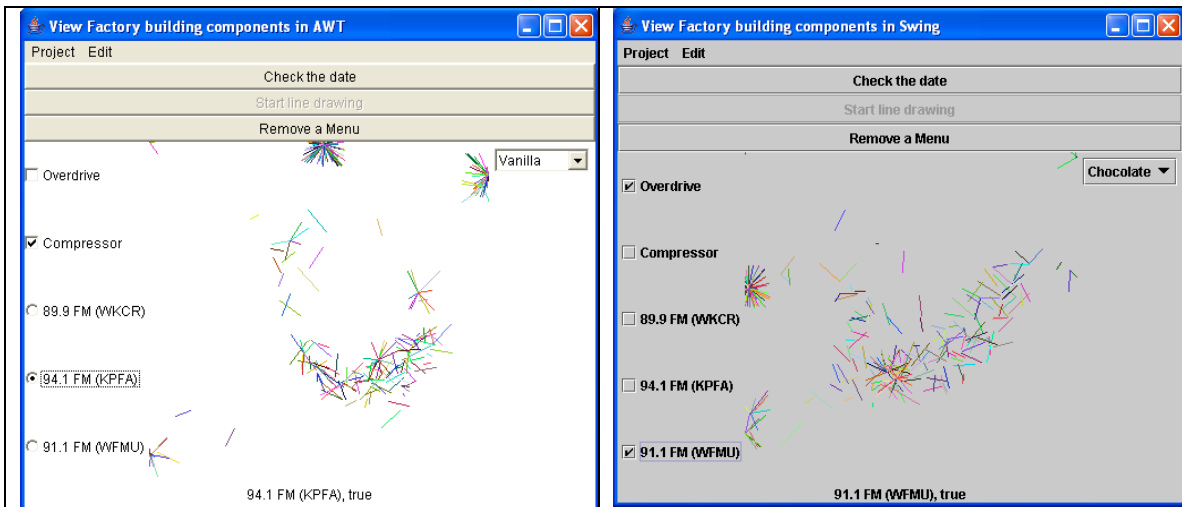
        myButton.addActionListener(this);
    }

    public void actionPerformed(ActionEvent ev) {
        Object source = ev.getSource();
        if (source == myButton) {
            String theDate = new java.util.Date().toString();
            myLabel.setText(theDate);
        }
    }

    public static void main(String[] args) {
        JMSL.setViewFactory(new ViewFactorySwing()); // change me
        SimpleFramePortView pvFrame = new SimpleFramePortView();
        pvFrame.pack();
        pvFrame.setVisible(true);
        pvFrame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}

```

The best way to learn more about this package is to read the Java documentation and more importantly, study the source code of `jmsltestsuite.PortViewTest`. Shown below is the frame built by this class, AWT on the left, Swing on the right.



HOMEWORK

Compose a piece in common music notation using custom JSyn sounds.

Design a `SynthNote` in `Wire` that has the required amplitude and frequency ports, but has additional input ports that give you more timbral control (the `halfLife` of an `ExponentialLag` for

example, or the cutoff frequency of a filter...).

Design a signal processing SynthNote in Wire (you must provide a SynthInput called "input"). Expose its control ports as well.

Export these as Java source from Wire, and compile them.

Compose a piece with your new instruments any way you like. You may do so by hand or algorithmically. Save the score as a zip file (just type .zip at the end of the filename when you save from JScore). Upload it to your website and use JMSLScoreApplet to deliver it. See the applet tag below. *Note that jmsl.jar, jscore.jar, and the directory containing your piece all must be in the classes directory. Note also that you must upload your SynthNotes to your classes directory as well.*

```
<applet code ="com.softsynth.jmsl.score.util.JMSLScoreApplet.class"
  codebase="classes"
  archive="jmsl.jar,jscore.jar"
  width="600"
  HEIGHT="129">
<param name ="URL" value ="JMSLScores/MyPiece.zip">
</applet>
```

Note again the "URL" param above. This assumes that there is a directory called **JMSLScores** in your **classes** directory, and you have placed your score file, named **MyPiece.zip** within it!